www.konkur.in

This International Student Edition is for use outside of the U.S.

SEVENTH EDITION

Database System Concepts



Abraham Silberschatz
Henry F. Korth
S. Sudarshan

Mc Graw Hill Education

Telegram: @uni_k

www.konkur.in

DATABASE
SYSTEM CONCEPTS

SEVENTH EDITION

Abraham Silberschatz
Yale University

Henry F. Korth
Lehigh University

S. Sudarshan
Indian Institute of Technology, Bombay

Mc Graw Hill Education

Telegram: @uni_k

www.konkur.in



DATABASE SYSTEM CONCEPTS

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121. Copyright © 2020 by McGraw-Hill Education. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw-Hill Education, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 21 20 19

ISBN 978-1-260-08450-4
MHID 1-260-08450-7

Cover Image: ©*Pavel Nesvadba/Shutterstock*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw-Hill Education, and McGraw-Hill Education does not guarantee the accuracy of the information presented at these sites.

mheducation.com/highered

Telegram: @uni_k

www.konkur.in

To meine schatzi, Valerie
her parents and my dear friends, Steve and Mary Anne
and in memory of my parents, Joseph and Vera

Avi Silberschatz

To my wife, Joan
my children, Abigail and Joseph
my mother, Frances
and in memory of my father, Henry

Hank Korth

To my wife, Sita
my children, Madhur and Advaith
and my mother, Indira

S. Sudarshan

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

About the Authors

Abraham (Avi) Silberschatz is the Sidney J. Weinberg Professor of Computer Science at Yale University. Prior to coming to Yale in 2003, he was the vice president of the Information Sciences Research Center at Bell Labs. He previously held an endowed professorship at the University of Texas at Austin, where he taught until 1993. Silberschatz is a fellow of the ACM, a fellow of the IEEE, and a member of the Connecticut Academy of Science and Engineering. He received the 2002 IEEE Taylor L. Booth Education Award, the 1998 ACM Karl V. Karlstrom Outstanding Educator Award, and the 1997 ACM SIGMOD Contribution Award. Silberschatz was awarded the Bell Laboratories President's Award three times, in 1998, 1999 and 2004. His writings have appeared in numerous journals, conferences, workshops, and book chapters. He has obtained over 48 patents and over 24 grants. He is an author of the textbook *Operating System Concepts*.

Henry F. (Hank) Korth is a Professor of Computer Science and Engineering and co-director of the Computer Science and Business program at Lehigh University. Prior to joining Lehigh, he was director of Database Principles Research at Bell Labs, a vice president of Panasonic Technologies, an associate professor at the University of Texas at Austin, and a research staff member at IBM Research. Korth is a fellow of the ACM and of the IEEE and a winner of the 10-Year Award at the VLDB Conference. His numerous research publications span a wide range of aspects of database systems, including transaction management in parallel and distributed systems, real-time systems, query processing, and the influence on these areas from modern computing architectures. Most recently, his research has addressed issues in the application of blockchains in enterprise databases.

S. Sudarshan is currently the Subrao M. Nilekani Chair Professor at the Indian Institute of Technology, Bombay. He received his Ph.D. at the University of Wisconsin in 1992, and he was a member of the technical staff at Bell Labs before joining IIT Bombay. Sudarshan is a fellow of the ACM. His research spans several areas of database systems, with a focus on query processing and query optimization. His paper on keyword search in databases published in 2002 won the IEEE ICDE Most Influential Paper Award in 2012, and his work on main-memory databases received the Bell Laboratories President's Award in 1999. His current research areas include testing and grading of SQL queries, optimization of database applications by rewriting of imperative code, and query optimization for parallel databases. He has published over 100 papers and obtained 15 patents.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

Contents

Chapter 1 Introduction

1.1 Database-System Applications 1
1.2 Purpose of Database Systems 5
1.3 View of Data 8
1.4 Database Languages 13
1.5 Database Design 17
1.6 Database Engine 18
1.7 Database and Application Architecture 21
1.8 Database Users and Administrators 24
1.9 History of Database Systems 25
1.10 Summary 29
Exercises 31
Further Reading 33

PART ONE ■ RELATIONAL LANGUAGES

Chapter 2 Introduction to the Relational Model

2.1 Structure of Relational Databases 37
2.2 Database Schema 41
2.3 Keys 43
2.4 Schema Diagrams 46
2.5 Relational Query Languages 47
2.6 The Relational Algebra 48
2.7 Summary 58
Exercises 60
Further Reading 63

Chapter 3 Introduction to SQL

3.1 Overview of the SQL Query Language 65
3.2 SQL Data Definition 66
3.3 Basic Structure of SQL Queries 71
3.4 Additional Basic Operations 79
3.5 Set Operations 85
3.6 Null Values 89
3.7 Aggregate Functions 91
3.8 Nested Subqueries 98
3.9 Modification of the Database 108
3.10 Summary 114
Exercises 115
Further Reading 124

Mc Graw Hill Education
5011111
Telegram: @uni_k

www.konkur.in

Chapter 4 Intermediate SQL

4.1 Join Expressions 125
4.2 Views 137
4.3 Transactions 143
4.4 Integrity Constraints 145
4.5 SQL Data Types and Schemas 153
4.6 Index Definition in SQL 164
4.7 Authorization 165
4.8 Summary 173
Exercises 176
Further Reading 180

Chapter 5 Advanced SQL

5.1 Accessing SQL from a Programming Language 183
5.2 Functions and Procedures 198
5.3 Triggers 206
5.4 Recursive Queries 213
5.5 Advanced Aggregation Features 219
5.6 Summary 231
Exercises 232
Further Reading 238

PART TWO ■ DATABASE DESIGN

Chapter 6 Database Design Using the E-R Model

6.1 Overview of the Design Process 241
6.2 The Entity-Relationship Model 244
6.3 Complex Attributes 249
6.4 Mapping Cardinalities 252
6.5 Primary Key 256
6.6 Removing Redundant Attributes in Entity Sets 261
6.7 Reducing E-R Diagrams to Relational Schemas 264
6.8 Extended E-R Features 271
6.9 Entity-Relationship Design Issues 279
6.10 Alternative Notations for Modeling Data 285
6.11 Other Aspects of Database Design 291
6.12 Summary 292
Exercises 294
Further Reading 300

Chapter 7 Relational Database Design

7.1 Features of Good Relational Designs 303
7.2 Decomposition Using Functional Dependencies 308
7.3 Normal Forms 313
7.4 Functional-Dependency Theory 320
7.5 Algorithms for Decomposition Using Functional Dependencies 330
7.6 Decomposition Using Multivalued Dependencies 336
7.7 More Normal Forms 341
7.8 Atomic Domains and First Normal Form 342
7.9 Database-Design Process 343
7.10 Modeling Temporal Data 347
7.11 Summary 351
Exercises 353
Further Reading 360

Telegram: @uni_k

www.konkur.in

PART THREE ■ APPLICATION DESIGN AND DEVELOPMENT

Chapter 8 Complex Data Types

8.1 Semi-structured Data 365
8.2 Object Orientation 376
8.3 Textual Data 382
8.4 Spatial Data 387
8.5 Summary 394
Exercises 397
Further Reading 401

Chapter 9 Application Development

9.1 Application Programs and User Interfaces 403
9.2 Web Fundamentals 405
9.3 Servlets 411
9.4 Alternative Server-Side Frameworks 416
9.5 Client-Side Code and Web Services 421
9.6 Application Architectures 429
9.7 Application Performance 434
9.8 Application Security 437
9.9 Encryption and Its Applications 447
9.10 Summary 453
Exercises 455
Further Reading 462

PART FOUR ■ BIG DATA ANALYTICS

Chapter 10 Big Data

10.1 Motivation 467
10.2 Big Data Storage Systems 472
10.3 The MapReduce Paradigm 483
10.4 Beyond MapReduce: Algebraic Operations 494
10.5 Streaming Data 500
10.6 Graph Databases 508
10.7 Summary 511
Exercises 513
Further Reading 516

Chapter 11 Data Analytics

11.1 Overview of Analytics 519
11.2 Data Warehousing 521
11.3 Online Analytical Processing 527
11.4 Data Mining 540
11.5 Summary 550
Exercises 552
Further Reading 555

Telegram: @uni_k

www.konkur.in

PART FIVE ■ STORAGE MANAGEMENT AND INDEXING

Chapter 12 Physical Storage Systems

12.1 Overview of Physical Storage Media 559
12.2 Storage Interfaces 562
12.3 Magnetic Disks 563
12.4 Flash Memory 567
12.5 RAID 570
12.6 Disk-Block Access 577
12.7 Summary 580
Exercises 582
Further Reading 584

Chapter 13 Data Storage Structures

13.1 Database Storage Architecture 587
13.2 File Organization 588
13.3 Organization of Records in Files 595
13.4 Data-Dictionary Storage 602
13.5 Database Buffer 604
13.6 Column-Oriented Storage 611
13.7 Storage Organization in Main-Memory Databases 615
13.8 Summary 617
Exercises 619
Further Reading 621

Chapter 14 Indexing

14.1 Basic Concepts 623
14.2 Ordered Indices 625
14.3 B^+-Tree Index Files 634
14.4 B^+-Tree Extensions 650
14.5 Hash Indices 658
14.6 Multiple-Key Access 661
14.7 Creation of Indices 664
14.8 Write-Optimized Index Structures 665
14.9 Bitmap Indices 670
14.10 Indexing of Spatial and Temporal Data 672
14.11 Summary 677
Exercises 679
Further Reading 683

PART SIX ■ QUERY PROCESSING AND OPTIMIZATION

Chapter 15 Query Processing

15.1 Overview 689
15.2 Measures of Query Cost 692
15.3 Selection Operation 695
15.4 Sorting 701
15.5 Join Operation 704
15.6 Other Operations 719
15.7 Evaluation of Expressions 724
15.8 Query Processing in Memory 731
15.9 Summary 734
Exercises 736
Further Reading 740

Telegram: @uni_k

www.konkur.in

Chapter 16 Query Optimization

16.1 Overview 743
16.2 Transformation of Relational Expressions 747
16.3 Estimating Statistics of Expression Results 757
16.4 Choice of Evaluation Plans 766
16.5 Materialized Views 778
16.6 Advanced Topics in Query Optimization 783
16.7 Summary 787
Exercises 789
Further Reading 794

PART SEVEN ■ TRANSACTION MANAGEMENT

Chapter 17 Transactions

17.1 Transaction Concept 799
17.2 A Simple Transaction Model 801
17.3 Storage Structure 804
17.4 Transaction Atomicity and Durability 805
17.5 Transaction Isolation 807
17.6 Serializability 812
17.7 Transaction Isolation and Atomicity 819
17.8 Transaction Isolation Levels 821
17.9 Implementation of Isolation Levels 823
17.10 Transactions as SQL Statements 826
17.11 Summary 828
Exercises 831
Further Reading 834

Chapter 18 Concurrency Control

18.1 Lock-Based Protocols 835
18.2 Deadlock Handling 849
18.3 Multiple Granularity 853
18.4 Insert Operations, Delete Operations, and Predicate Reads 857
18.5 Timestamp-Based Protocols 861
18.6 Validation-Based Protocols 866
18.7 Multiversion Schemes 869
18.8 Snapshot Isolation 872
18.9 Weak Levels of Consistency in Practice 880
18.10 Advanced Topics in Concurrency Control 883
18.11 Summary 894
Exercises 899
Further Reading 904

Chapter 19 Recovery System

19.1 Failure Classification 907
19.2 Storage 908
19.3 Recovery and Atomicity 912
19.4 Recovery Algorithm 922
19.5 Buffer Management 926
19.6 Failure with Loss of Non-Volatile Storage 930
19.7 High Availability Using Remote Backup Systems 931
19.8 Early Lock Release and Logical Undo Operations 935
19.9 ARIES 941
19.10 Recovery in Main-Memory Databases 947
19.11 Summary 948
Exercises 952
Further Reading 956

Telegram: @uni_k

www.konkur.in

PART EIGHT ■ PARALLEL AND DISTRIBUTED DATABASES

Chapter 20 Database-System Architectures

20.1 Overview 961
20.2 Centralized Database Systems 962
20.3 Server System Architectures 963
20.4 Parallel Systems 970
20.5 Distributed Systems 986
20.6 Transaction Processing in Parallel and Distributed Systems 989
20.7 Cloud-Based Services 990
20.8 Summary 995
Exercises 998
Further Reading 1001

Chapter 21 Parallel and Distributed Storage

21.1 Overview 1003
21.2 Data Partitioning 1004
21.3 Dealing with Skew in Partitioning 1007
21.4 Replication 1013
21.5 Parallel Indexing 1017
21.6 Distributed File Systems 1019
21.7 Parallel Key-Value Stores 1023
21.8 Summary 1032
Exercises 1033
Further Reading 1036

Chapter 22 Parallel and Distributed Query Processing

22.1 Overview 1039
22.2 Parallel Sort 1041
22.3 Parallel Join 1043
22.4 Other Operations 1048
22.5 Parallel Evaluation of Query Plans 1052
22.6 Query Processing on Shared-Memory Architectures 1061
22.7 Query Optimization for Parallel Execution 1064
22.8 Parallel Processing of Streaming Data 1070
22.9 Distributed Query Processing 1076
22.10 Summary 1086
Exercises 1089
Further Reading 1093

Chapter 23 Parallel and Distributed Transaction Processing

23.1 Distributed Transactions 1098
23.2 Commit Protocols 1100
23.3 Concurrency Control in Distributed Databases 1111
23.4 Replication 1121
23.5 Extended Concurrency Control Protocols 1129
23.6 Replication with Weak Degrees of Consistency 1133
23.7 Coordinator Selection 1146
23.8 Consensus in Distributed Systems 1150
23.9 Summary 1162
Exercises 1165
Further Reading 1168

Telegram: @uni_k

www.konkur.in

PART NINE ■ ADVANCED TOPICS

Chapter 24 Advanced Indexing Techniques

24.1 Bloom Filter 1175
24.2 Log-Structured Merge Tree and Variants 1176
24.3 Bitmap Indices 1182
24.4 Indexing of Spatial Data 1186
24.5 Hash Indices 1190
24.6 Summary 1203
Exercises 1205
Further Reading 1206

Chapter 25 Advanced Application Development

25.1 Performance Tuning 1210
25.2 Performance Benchmarks 1230
25.3 Other Issues in Application Development 1234
25.4 Standardization 1237
25.5 Distributed Directory Systems 1240
25.6 Summary 1243
Exercises 1245
Further Reading 1248

Chapter 26 Blockchain Databases

26.1 Overview 1252
26.2 Blockchain Properties 1254
26.3 Achieving Blockchain Properties via Cryptographic Hash Functions 1259
26.4 Consensus 1263
26.5 Data Management in a Blockchain 1267
26.6 Smart Contracts 1269
26.7 Performance Enhancement 1274
26.8 Emerging Applications 1276
26.9 Summary 1279
Exercises 1280
Further Reading 1282

PART TEN ■ APPENDIX A

Appendix A Detailed University Schema 1287

Index 1299

PART ELEVEN ■ ONLINE CHAPTERS

Chapter 27 Formal Relational Query Languages
Chapter 28 Advanced Relational Database Design
Chapter 29 Object-Based Databases
Chapter 30 XML
Chapter 31 Information Retrieval
Chapter 32 PostgreSQL

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

Preface

Database management has evolved from a specialized computer application to a central component of virtually all enterprises, and, as a result, knowledge about database systems has become an essential part of an education in computer science. In this text, we present the fundamental concepts of database management. These concepts include aspects of database design, database languages, and database-system implementation.

This text is intended for a first course in databases at the junior or senior undergraduate, or first-year graduate, level. In addition to basic material for a first course, the text contains advanced material that can be used for course supplements, or as introductory material for an advanced course.

We assume only a familiarity with basic data structures, computer organization, and a high-level programming language such as Java, C, C++, or Python. We present concepts as intuitive descriptions, many of which are based on our running example of a university. Important theoretical results are covered, but formal proofs are omitted. In place of proofs, figures and examples are used to suggest why a result is true. Formal descriptions and proofs of theoretical results may be found in research papers and advanced texts that are referenced in the bibliographical notes.

The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial or experimental database systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular database system, though we do provide references to specific systems where appropriate.

In this, the seventh edition of *Database System Concepts*, we have retained the overall style of the prior editions while evolving the content and organization to reflect the changes that are occurring in the way databases are designed, managed, and used. One such major change is the extensive use of "Big Data" systems. We have also taken into account trends in the teaching of database concepts and made adaptations to facilitate these trends where appropriate.

Telegram: @uni_k

www.konkur.in

Among the notable changes in this edition are:

- Extensive coverage of Big Data systems, from the user perspective (Chapter 10), as well as from an internal perspective (Chapter 20 through Chapter 23), with extensive additions and modifications compared to the sixth edition.
- A new chapter entitled "Blockchain Databases" (Chapter 26) that introduces blockchain technology and its growing role in enterprise applications. An important focus in this chapter is the interaction between blockchain systems and database systems.
- Updates to all chapters covering database internals (Chapter 12 through Chapter 19) to reflect current-generation technology, such as solid-state disks, main-memory databases, multi-core systems, and column-stores.
- Enhanced coverage of semi-structured data management using JSON, RDF, and SPARQL (Section 8.1).
- Updated coverage of temporal data (in Section 7.10), data analytics (Chapter 11), and advanced indexing techniques such as write-optimized indices (Section 14.8 and Section 24.2).
- Reorganization and update of chapters to better support courses with a significant hands-on component (which we strongly recommend for any database course), including use of current-generation application development tools and Big Data systems such as Apache Hadoop and Spark.

These and other updates have arisen from the many comments and suggestions we have received from readers of the sixth edition, our students at Yale University, Lehigh University, and IIT Bombay, and our own observations and analyses of developments in database technology.

Content of This Book

The text is organized in eleven major parts.

- **Overview** (Chapter 1). Chapter 1 provides a general overview of the nature and purpose of database systems. We explain how the concept of a database system has developed, what the common features of database systems are, what a database system does for the user, and how a database system interfaces with operating systems. We also introduce an example database application: a university organization consisting of multiple departments, instructors, students, and courses. This application is used as a running example throughout the book. This chapter is motivational, historical, and explanatory in nature.

Telegram: @uni_k

www.konkur.in

- **Part 1: Relational Model and SQL** (Chapter 2 through Chapter 5). Chapter 2 introduces the relational model of data, covering basic concepts such as the structure of relational databases, database schemas, keys, schema diagrams, relational query languages, relational operations, and the relational algebra. Chapter 3, Chapter 4, and Chapter 5 focus on the most influential of the user-oriented relational languages: SQL. The chapters in this part describe data manipulation: queries, updates, insertions, and deletions, assuming a schema design has been provided. Although data-definition syntax is covered in detail here, schema design issues are deferred to Part 2.
- **Part 2: Database Design** (Chapter 6 and Chapter 7). Chapter 6 provides an overview of the database-design process and a detailed description of the entity-relationship data model. The entity-relationship data model provides a high-level view of the issues in database design and of the problems encountered in capturing the semantics of realistic applications within the constraints of a data model. UML class-diagram notation is also covered in this chapter. Chapter 7 introduces relational database design. The theory of functional dependencies and normalization is covered, with emphasis on the motivation and intuitive understanding of each normal form. This chapter begins with an overview of relational design and relies on an intuitive understanding of logical implication of functional dependencies. This allows the concept of normalization to be introduced prior to full coverage of functional-dependency theory, which is presented later in the chapter. Instructors may choose to use only this initial coverage without loss of continuity. Instructors covering the entire chapter will benefit from students having a good understanding of normalization concepts to motivate them to learn some of the challenging concepts of functional-dependency theory. The chapter ends with a section on modeling of temporal data.
- **Part 3: Application Design and Development** (Chapter 8 and Chapter 9). Chapter 8 discusses several complex data types that are particularly important for application design and development, including semi-structured data, object-based data, textual data, and spatial data. Although the popularity of XML in a database context has been diminishing, we retain an introduction to XML, while adding coverage of JSON, RDF, and SPARQL. Chapter 9 discusses tools and technologies that are used to build interactive web-based and mobile database applications. This chapter includes detailed coverage on both the server side and the client side. Among the topics covered are servlets, JSP, Django, JavaScript, and web services. Also discussed are application architecture, object-relational mapping systems including Hibernate and Django, performance (including caching using memcached and Redis), and the unique challenges in ensuring web-application security.
- **Part 4: Big Data Analytics** (Chapter 10 and Chapter 11). Chapter 10 provides an overview of large-scale data-analytic applications, with a focus on how those applications place distinct demands on data management compared with the de-

Telegram: @uni_k

www.konkur.in

mands of traditional database applications. The chapter then discusses how those demands are addressed. Among the topics covered are Big Data storage systems including distributed file systems, key-value stores and NoSQL systems, MapReduce, Apache Spark, streaming data, and graph databases. The connection of these systems and concepts with database concepts introduced earlier is emphasized. Chapter 11 discusses the structure and use of systems designed for large-scale data analysis. After first explaining the concepts of data analytics, business intelligence, and decision support, the chapter discusses the structure of a data warehouse and the process of gathering data into a warehouse. The chapter next covers usage of warehouse data in OLAP applications followed by a survey of data-mining algorithms and techniques.

- **Part 5: Storage Management and Indexing** (Chapter 12 through Chapter 14). Chapter 12 deals with storage devices and how the properties of those devices influence database physical organization and performance. Chapter 13 deals with data-storage structures, including file organization and buffer management. A variety of data-access techniques are presented in Chapter 14. Multilevel index-based access is described, culminating in detailed coverage of B^+-trees. The chapter then covers index structures for applications where the B^+-tree structure is less appropriate, including write-optimized indices such as LSM trees and buffer trees, bitmap indices, and the indexing of spatial data using k-d trees, quadtrees and R-trees.
- **Part 6: Query Processing and Optimization** (Chapter 15 and Chapter 16). Chapter 15 and Chapter 16 address query-evaluation algorithms and query optimization. Chapter 15 focuses on algorithms for the implementation of database operations, particularly the wide range of join algorithms, which are designed to work on very large data that may not fit in main-memory. Query processing techniques for main-memory databases are also covered in this chapter. Chapter 16 covers query optimization, starting by showing how query plans can be transformed to other equivalent plans by using transformation rules. The chapter then describes how to generate estimates of query execution costs, and how to efficiently find query execution plans with the lowest cost.
- **Part 7: Transaction Management** (Chapter 17 through Chapter 19). Chapter 17 focuses on the fundamentals of a transaction-processing system: atomicity, consistency, isolation, and durability. It provides an overview of the methods used to ensure these properties, including log-based recovery and concurrency control using locking, timestamp-based techniques, and snapshot isolation. Courses requiring only a survey of the transaction concept can use Chapter 17 on its own without the other chapters in this part; those chapters provide significantly greater depth. Chapter 18 focuses on concurrency control and presents several techniques for ensuring serializability, including locking, timestamping, and optimistic (validation) techniques. Multiversion concurrency control techniques, including the widely used snapshot isolation technique, and an extension of the technique that

Telegram: @uni_k

www.konkur.in

guarantees serializability, are also covered. This chapter also includes discussion of weak levels of consistency, concurrency on index structures, concurrency in main-memory database systems, long-duration transactions, operation-level concurrency, and real-time transaction processing. Chapter 19 covers the primary techniques for ensuring correct transaction execution despite system crashes and storage failures. These techniques include logs, checkpoints, and database dumps, as well as high availability using remote backup systems. Recovery with early lock release, and the widely used ARIES algorithm are also presented. This chapter includes discussion of recovery in main-memory database systems and the use of NVRAM.

- **Part 8: Parallel and Distributed Databases** (Chapter 20 through Chapter 23). Chapter 20 covers computer-system architecture, and describes the influence of the underlying computer system on the database system. We discuss centralized systems, client–server systems, parallel and distributed architectures, and cloud-based systems in this chapter. The remaining three chapters in this part address distinct aspects of parallel and distributed databases, with Chapter 21 covering storage and indexing, Chapter 22 covering query processing, and Chapter 23 covering transaction management. Chapter 21 includes discussion of partitioning and data skew, replication, parallel indexing, distributed file systems (including the Hadoop file system), and parallel key-value stores. Chapter 22 includes discussion of parallelism both among multiple queries and within a single query. It covers parallel and distributed sort and join, MapReduce, pipelining, the Volcano exchange-operator model, thread-level parallelism, streaming data, and the optimization of geographically distributed queries. Chapter 23 includes discussion of traditional distributed consensus such as two-phase commit and more sophisticated solutions including Paxos and Raft. It covers a variety of algorithms for distributed concurrency control, including replica management and weaker degrees of consistency. The trade-offs implied by the CAP theorem are discussed along with the means of detecting inconsistency using version vectors and Merkle trees.
- **Part 9: Advanced Topics** (Chapter 24 through Chapter 26). Chapter 24 expands upon the coverage of indexing in Chapter 14 with detailed coverage of the LSM tree and its variants, bitmap indices, spatial indexing, and dynamic hashing techniques. Chapter 25 expands upon the coverage of Chapter 9 with a discussion of performance tuning, benchmarking, testing, and migration from legacy systems, standardization, and distributed directory systems. Chapter 26 looks at blockchain technology from a database perspective. It describes blockchain data structures and the use of cryptographic hash functions and public-key encryption to ensure the blockchain properties of anonymity, irrefutability, and tamper resistance. It describes and compares the distributed consensus algorithms used to ensure decentralization, including proof-of-work, proof-of-stake, and Byzantine consensus. Much of the chapter focuses on the features that make blockchain an important database concept, including the role of permisssioned blockchains, the encoding

Telegram: @uni_k

www.konkur.in

of business logic and agreements in smart contracts, and interoperability across blockchains. Techniques for achieving database-scale transaction-processing performance are discussed. A final section surveys current and contemplated enterprise blockchain applications.

- **Part 10: Appendix**. Appendix A presents details of our university schema, including the full schema, DDL, and all the tables.
- **Part 11: Online Chapters** (Chapter 27 through Chapter 32) available online at db-book.com. We provide six chapters that cover material that is of historical nature or is advanced; these chapters are available only online. Chapter 27 covers "pure" query languages: the tuple and domain relational calculus and Datalog, which has a syntax modeled after the Prolog language. Chapter 28 covers advanced topics in relational database design, including the theory of multivalued dependencies and fourth normal form, as well as higher normal forms. Chapter 29 covers object-based databases and more complex data types such as array, and multiset types, as well as tables that are not in 1NF. Chapter 30 expands on the coverage in Chapter 8 of XML. Chapter 31 covers information retrieval, which deals with querying of unstructured textual data. Chapter 32 provides an overview of the PostgreSQL database system, and is targeted at courses focusing on database internals. The chapter is likely to be particularly useful for supporting student projects that work with the open-source code base of the PostgreSQL database.

At the end of each chapter we provide references in a section titled *Further Reading*. This section is intentionally abbreviated and provides references that allow students to continue their study of the material covered in the chapter or to learn about new developments in the area covered by the chapter. On occasion, the further reading section includes original source papers that have become classics of which everyone should be aware. Detailed bibliographical notes for each chapter are available online, and provide references for readers who wish to go into further depth on any of the topics covered in the chapter.

The Seventh Edition

The production of this seventh edition has been guided by the many comments and suggestions we received concerning the earlier editions, by our own observations while teaching at Yale University, Lehigh University, and IIT Bombay, and by our analysis of the directions in which database technology is evolving.

We provided a list of the major new features of this edition earlier in this preface; these include coverage of extensive coverage of Big Data, updates to all chapters to reflect current generation hardware technology, semi-structured data management, advanced indexing techniques, and a new chapter on blockchain databases. Beyond these major changes, we revised the material in each chapter, bringing the older material

Telegram: @uni_k

www.konkur.in

up-to-date, adding discussions on recent developments in database technology, and improving descriptions of topics that students found difficult to understand. We have also added new exercises and updated references.

For instructors who previously used the sixth edition, we list the more significant changes below:

- Relational algebra has been moved into Chapter 2, to help students better understand relational operations that form the basis of query languages such as SQL. Deeper coverage of relational algebra also aids in understanding the algebraic operators needed for discussion later of query processing and optimization. The two variants of the relational calculus are now in an online chapter, since we believe they are now of value only to more theoretically oriented courses, and can be omitted by most database courses.
- The SQL chapters now include more details of database-system specific SQL variations, to aid students carrying out practical assignments. Connections between SQL and the multiset relational algebra are also covered in more detail. Chapter 4 now covers all the material concerning joins, whereas previously natural join was in the preceding chapter. Coverage of sequences used to generate unique key values, and coverage of row-level security have also been added to this chapter. Recent extensions to the JDBC API that are particularly useful are now covered in Chapter 5; coverage of OLAP has been moved from this chapter to Chapter 11.
- Chapter 6 has been modified to cover E-R diagrams along with E-R concepts, instead of first covering the concepts and then introducing E-R diagrams as was done in earlier editions. We believe this will help students better comprehend the E-R model.
- Chapter 7 now has improved coverage of temporal data modeling, including SQL:2011 temporal database features.
- Chapter 8 is a new chapter that covers complex data types, including semi-structured data, such as XML, JSON, RDF, and SPARQL, object-based data, textual data, and spatial data. Object-based databases, XML, and information retrieval on textual data were covered in detail in the sixth edition; these topics have been abbreviated and covered in Chapter 8, while the original chapters from the sixth edition have now been made available online.
- Chapter 9 has been significantly updated to reflect modern application development tools and techniques, including extended coverage of JavaScript and JavaScript libraries for building dynamic web interfaces, application development in Python using the Django framework, coverage of web services, and disconnection operations using HTML5. Object-relation mapping using Django has been added, as also discussion of techniques for developing high-performance applications that can handle large transaction loads.

Telegram: @uni_k

www.konkur.in

- Chapter 10 is a new chapter on Big Data, covering Big Data concepts and tools from a user perspective. Big Data storage systems, the MapReduce paradigm, Apache Hadoop and Apache Spark, and streaming and graph databases are covered in this chapter. The goal is to enable readers to use Big Data systems, with only a summary coverage of what happens behind the scenes. Big Data internals are covered in detail in later chapters.
- The chapter on storage and file structure has been split into two chapters. Chapter 12 which covers storage has been updated with new technology, including expanded coverage of flash memory, column-oriented storage, and storage organization in main-memory databases. Chapter 13, which covers data storage structures has been expanded, and now covers details such as free-space maps, partitioning, and most importantly column-oriented storage.
- Chapter 14 on indexing now covers write-optimized index structures including the LSM tree and its variants, and the buffer tree, which are seeing increasing usage. Spatial indices are now covered briefly in this chapter. More detailed coverage of LSM trees and spatial indices is provided in Chapter 24, which covers advanced indexing techniques. Bitmap indices are now covered in brief in Chapter 14, while more detailed coverage has been moved to Chapter 24. Dynamic hashing techniques have been moved into Chapter 24, since they are of limited practical importance today.
- Chapter 15 on query processing has significantly expanded coverage of pipelining in query processing, new material on query processing in main-memory, including query compilation, as well as brief coverage of spatial joins. Chapter 16 on query optimization has more examples of equivalence rules for operators such as outer joins and aggregates, has updated material on statistics for cost estimation, an improved presentation of the join-order optimization algorithm. Techniques for decorrelating nested subqueries using semijoin and antijoin operations have also been added.
- Chapter 18 on concurrency control has new material on concurrency control in main-memory. Chapter 19 on recovery now gives more importance to high availability using remote backup systems.
- Our coverage of parallel and distributed databases has been completely revamped. Because of the evolution of these two areas into a continuum from low-level parallelism to geographically distributed systems, we now present these topics together.
    - Chapter 20 on database architectures has been significantly updated from the earlier edition, including new material on practical interconnection networks like the tree-like (or fat-tree) architecture, and significantly expanded and updated material on shared-memory architectures and cache coherency. There is an entirely new section on cloud-based services, covering virtual machines and containers, platform-as-a-service, software-as-a-service, and elasticity.

Telegram: @uni_k

www.konkur.in

- Chapter 21 covers parallel and distributed storage; while a few parts of this chapter were present in the sixth edition, such as partitioning techniques, everything else in this chapter is new.
- Chapter 22 covers parallel and distributed query processing. Again only a few sections of this chapter, such as parallel algorithms for sorting, join, and a few other relational operations, were present in the sixth edition, almost everything else in this chapter is new.
- Chapter 23 covers parallel and distributed transaction processing. A few parts of this chapter, such as the sections on 2PC, persistent messaging, and concurrency control in distributed databases, are new but almost everything else in this chapter is new.

As in the sixth edition, we facilitate the following of our running example by listing the database schema and the sample relation instances for our university database together in Appendix A as well as where they are used in the various regular chapters. In addition, we provide, on our web site db-book.com, SQL data-definition statements for the entire example, along with SQL statements to create our example relation instances. This encourages students to run example queries directly on a database system and to experiment with modifying those queries. All topics not listed above are updated from the sixth edition, though their overall organization is relatively unchanged.

End of Chapter Material

Each chapter has a list of review terms, in addition to a summary, which can help readers review key topics covered in the chapter.

As in the sixth edition, the exercises are divided into two sets: **practice exercises** and **exercises**. The solutions for the practice exercises are publicly available on the web site of the book. Students are encouraged to solve the practice exercises on their own and later use the solutions on the web site to check their own solutions. Solutions to the other exercises are available only to instructors (see "Instructor's Note," below, for information on how to get the solutions).

Many chapters have a tools section at the end of the chapter that provides information on software tools related to the topic of the chapter; some of these tools can be used for laboratory exercises. SQL DDL and sample data for the university database and other relations used in the exercises are available on the web site of the book and can be used for laboratory exercises.

Instructor's Note

It is possible to design courses by using various subsets of the chapters. Some of the chapters can also be covered in an order different from their order in the book. We outline some of the possibilities here:

Telegram: @uni_k

www.konkur.in

- Chapter 5 (Advanced SQL). This chapter can be skipped or deferred to later without loss of continuity. We expect most courses will cover at least Section 5.1.1 early, as JDBC is likely to be a useful tool in student projects.
- Chapter 6 (E-R Model). This chapter can be covered ahead of Chapter 3, Chapter 4, and Chapter 5 if you so desire, since Chapter 6 does not have any dependency on SQL. However, for courses with a programming emphasis, a richer variety of laboratory exercises is possible after studying SQL, and we recommend that SQL be covered before database design for such courses.
- Chapter 15 (Query Processing) and Chapter 16 (Query Optimization). These chapters can be omitted from an introductory course without affecting coverage of any other chapter.
- Part 7 (Transaction Management). Our coverage consists of an overview (Chapter 17) followed by chapters with details. You might choose to use Chapter 17 while omitting Chapter 18 and Chapter 19, if you defer these latter chapters to an advanced course.
- Part 8 (Parallel and Distributed Databases). Our coverage consists of an overview (Chapter 20), followed by chapters on the topics of storage, query processing, and transactions. You might choose to use Chapter 20 while omitting Chapter 21 through Chapter 23 if you defer these latter chapters to an advanced course.
- Part 11 (Online chapters). Chapter 27 (Formal-Relational Query Languages). This chapter can be covered immediately after Chapter 2, ahead of SQL. Alternatively, this chapter may be omitted from an introductory course. The five other online chapters (Advanced Relational Database Design, Object-Based Databases, XML, Information Retrieval, and PostgreSQL) can be used as self-study material or omitted from an introductory course.

Model course syllabi, based on the text, can be found on the web site of the book.

Web Site and Teaching Supplements

A web site for the book is available at the URL: db-book.com. The web site contains:

- Slides covering all the chapters of the book.
- Answers to the practice exercises.
- The six online chapters.
- Laboratory material, including SQL DDL and sample data for the university schema and other relations used in exercises, and instructions for setting up and using various database systems and tools.
- An up-to-date errata list.

Telegram: @uni_k

www.konkur.in

The following additional material is available only to faculty:

- An instructor's manual containing solutions to all exercises in the book.
- A question bank containing extra exercises.

For more information about how to get a copy of the instructor's manual and the question bank, please send an email message to sem@mheducation.com. In the United States, you may call 800-338-3987. The McGraw-Hill web site for this book is www.mhhe.com/silberschatz.

Contacting Us

We have endeavored to eliminate typos, bugs, and the like from the text. But, as in new releases of software, bugs almost surely remain; an up-to-date errata list is accessible from the book's web site. We would appreciate it if you would notify us of any errors or omissions in the book that are not on the current list of errata.

We would be glad to receive suggestions on improvements to the book. We also welcome any contributions to the book web site that could be of use to other readers, such as programming exercises, project suggestions, online labs and tutorials, and teaching tips.

Email should be addressed to db-book-authors@cs.yale.edu. Any other correspondence should be sent to Avi Silberschatz, Department of Computer Science, Yale University, 51 Prospect Street, P.O. Box 208285, New Haven, CT 06520-8285 USA.

Acknowledgments

Many people have helped us with this seventh edition, as well as with the previous six editions from which it is derived, and we are indebted to all of them.

Seventh Edition

- Ioannis Alagiannis and Renata Borovica-Gajic for writing Chapter 32 on the PostgreSQL database, which is available online. The chapter is a complete rewrite of the PostgreSQL chapter in the 6th edition, which was authored by Anastasia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder, Karl Schnaitter, and Gavin Sherry.
- Judi Paige for her help in generating figures, presentation slides, and with handling the copy-editing material.
- Mark Wogahn for making sure that the software to produce the book, including LaTeX macros and fonts, worked properly.

Telegram: @uni_k

www.konkur.in

- Sriram Srinivasan for discussions and feedback that have immensely benefited the chapters on parallel and distributed databases.
- N. L. Sarda for his insightful feedback on the sixth edition, and on some sections of the seventh edition.
- Bikash Chandra and Venkatesh Emani for their help with updates to the application development chapter, including creation of sample code.
- Students at IIT Bombay, particularly Ashish Mithole, for their feedback on draft versions of the chapters on parallel and distributed databases.
- Students at Yale, Lehigh, and IIT Bombay, for their comments on the sixth edition.
- Jeffrey Anthony, partner and CTO, Synaptic; and Lehigh students Corey Caplan (now co-founder, Leavitt Innovations); Gregory Cheng; Timothy LaRowe; and Aaron Rotem for comments and suggestions that have benefited the new blockchain chapter.

Previous Editions

- Hakan Jakobsson (Oracle), for writing the chapter on the Oracle database system in the sixth edition; Sriram Padmanabhan (IBM), for writing the chapter describing the IBM DB2 database system in the sixth edition; and Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas, and Michael Zwilling for writing the chapter describing the Microsoft SQL Server database system in the sixth edition; and in particular José Blakeley, who sadly is no longer amongst us, for coordinating and editing the chapter; and César Galindo-Legaria, Goetz Graefe, Kalen Delaney, and Thomas Casey for their contributions to the previous edition of the Microsoft SQL Server chapter. These chapters, however, are not part of the seventh edition.
- Anastasia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder, Karl Schnaitter, and Gavin Sherry for writing the chapter on PostgreSQL in the sixth edition.
- Daniel Abadi for reviewing the table of contents of the fifth edition and helping with the new organization.
- Steve Dolins, University of Florida; Rolando Fernanez, George Washington University; Frantisek Franek, McMaster University; Latifur Khan, University of Texas at Dallas; Sanjay Madria, Missouri University of Science and Technology; Aris Ouksel, University of Illinois; and Richard Snodgrass, University of Waterloo; who served as reviewers of the book and whose comments helped us greatly in formulating the sixth edition.

Telegram: @uni_k

www.konkur.in

- Judi Paige for her help in generating figures and presentation slides.
- Mark Wogahn for making sure that the software to produce the book, including LaTeX macros and fonts, worked properly.
- N. L. Sarda for feedback that helped us improve several chapters. Vikram Pudi for motivating us to replace the earlier bank schema; and Shetal Shah for feedback on several chapters.
- Students at Yale, Lehigh, and IIT Bombay, for their comments on the fifth edition, as well as on preprints of the sixth edition.
- Chen Li and Sharad Mehrotra for providing material on JDBC and security for the fifth edition.
- Marilyn Turnamian and Nandprasad Joshi provided secretarial assistance for the fifth edition, and Marilyn also prepared an early draft of the cover design for the fifth edition.
- Lyn Dupré copyedited the third edition and Sara Strandtman edited the text of the third edition.
- Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K. V. Raghavan, Prateek Kapadia, Sara Strandtman, Greg Speegle, and Dawn Bezviner helped to prepare the instructor's manual for earlier editions.
- The idea of using ships as part of the cover concept was originally suggested to us by Bruce Stephan.
- The following people offered suggestions and comments for the fifth and earlier editions of the book. R. B. Abhyankar, Hani Abu-Salem, Jamel R. Alsabbagh, Raj Ashar, Don Batory, Phil Bernhard, Christian Breimann, Gavin M. Bierman, Janek Bogucki, Haran Boral, Paul Bourgeois, Phil Bohannon, Robert Brazile, Yuri Breitbart, Ramzi Bualuan, Michael Carey, Soumen Chakrabarti, Tom Chappell, Zhengxin Chen, Y. C. Chin, Jan Chomicki, Laurens Damen, Prasanna Dhandapani, Qin Ding, Valentin Dinu, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Frantisek Franek, Shashi Gadia, Hector Garcia-Molina, Goetz Graefe, Jim Gray, Le Gruenwald, Eitan M. Gurari, William Hankley, Bruce Hillyer, Ron Hitchens, Chad Hogg, Arvind Hulgeri, Yannis Ioannidis, Zheng Jiaping, Randy M. Kaplan, Graham J. L. Kemp, Rami Khouri, Hyoung-Joo Kim, Won Kim, Henry Korth (father of Henry F.), Carol Kroll, Hae Choon Lee, Sang-Won Lee, Irwin Levinstein, Mark Llewellyn, Gary Lindstrom, Ling Liu, Dave Maier, Keith Marzullo, Marty Maskarinec, Fletcher Mattox, Sharad Mehrotra, Jim Melton, Alberto Mendelzon, Ami Motro, Bhagirath Narahari, Yiu-Kai Dennis Ng, Thanh-Duy Nguyen, Anil Nigam, Cyril Orji, Meral Ozsoyoglu, D. B. Phatak, Juan Altmayer Pizzorno, Bruce Porter, Sunil Prabhakar, Jim Peterson, K. V. Raghavan, Nahid Rahman, Rajarshi Rakshit, Krithi Ramamritham, Mike Reiter, Greg Ric-

Telegram: @uni_k

www.konkur.in

cardi, Odinaldo Rodriguez, Mark Roth, Marek Rusinkiewicz, Michael Rys, Sunita Sarawagi, N. L. Sarda, Patrick Schmid, Nikhil Sethi, S. Seshadri, Stewart Shen, Shashi Shekhar, Amit Sheth, Max Smolens, Nandit Soparkar, Greg Speegle, Jeff Storey, Dilys Thomas, Prem Thomas, Tim Wahls, Anita Whitehall, Christopher Wilson, Marianne Winslett, Weining Zhang, and Liu Zhenming.

Personal Notes

Sudarshan would like to acknowledge his wife, Sita, for her love, patience, and support, and children Madhur and Advaith for their love and joie de vivre. Hank would like to acknowledge his wife, Joan, and his children, Abby and Joe, for their love and understanding. Avi would like to acknowledge Valerie for her love, patience, and support during the revision of this book.

A. S.
H. F. K.
S. S.

Telegram: @uni_k

www.konkur.in

CHAPTER 1



Introduction

A **database-management system** (**DBMS**) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and techniques form the focus of this book. This chapter briefly introduces the principles of database systems.

1.1 Database-System Applications

The earliest database systems arose in the 1960s in response to the computerized management of commercial data. Those earlier applications were relatively simple compared to modern database applications. Modern applications include highly sophisticated, worldwide enterprises.

All database applications, old and new, share important common elements. The central aspect of the application is not a program performing some calculation, but rather the data themselves. Today, some of the most valuable corporations are valuable not because of their physical assets, but rather because of the information they own. Imagine a bank without its data on accounts and customers or a social-network site that loses the connections among its users. Such companies' value would be almost totally lost under such circumstances.

Telegram: @uni_k

www.konkur.in

Database systems are used to manage collections of data that:

- are highly valuable,
- are relatively large, and
- are accessed by multiple users and applications, often at the same time.

The first database applications had only simple, precisely formatted, structured data. Today, database applications may include data with complex relationships and a more variable structure. As an example of an application with structured data, consider a university's records regarding courses, students, and course registration. The university keeps the same type of information about each course: course-identifier, title, department, course number, etc., and similarly for students: student-identifier, name, address, phone, etc. Course registration is a collection of pairs: one course identifier and one student identifier. Information of this sort has a standard, repeating structure and is representative of the type of database applications that go back to the 1960s. Contrast this simple university database application with a social-networking site. Users of the site post varying types of information about themselves ranging from simple items such as name or date of birth, to complex posts consisting of text, images, videos, and links to other users. There is only a limited amount of common structure among these data. Both of these applications, however, share the basic features of a database.

Modern database systems exploit commonalities in the structure of data to gain efficiency but also allow for weakly structured data and for data whose formats are highly variable. As a result, a database system is a large, complex software system whose task is to manage a large, complex collection of data.

Managing complexity is challenging, not only in the management of data but in any domain. Key to the management of complexity is the concept of *abstraction*. Abstraction allows a person to use a complex device or system without having to know the details of how that device or system is constructed. A person is able, for example, to drive a car by knowing how to operate its controls. However, the driver does not need to know how the motor was built nor how it operates. All the driver needs to know is an abstraction of what the motor does. Similarly, for a large, complex collection of data, a database system provides a simpler, abstract view of the information so that users and application programmers do not need to be aware of the underlying details of how data are stored and organized. By providing a high level of abstraction, a database system makes it possible for an enterprise to combine data of various types into a unified repository of the information needed to run the enterprise.

Here are some representative applications:

- **Enterprise Information**
  - **Sales:** For customer, product, and purchase information.

Telegram: @uni_k

www.konkur.in

- **Accounting:** For payments, receipts, account balances, assets, and other accounting information.
- **Human resources:** For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.

- **Manufacturing:** For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
- **Banking and Finance**
  - **Banking:** For customer information, accounts, loans, and banking transactions.
  - **Credit card transactions:** For purchases on credit cards and generation of monthly statements.
  - **Finance:** For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
- **Universities:** For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- **Airlines:** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- **Telecommunication:** For keeping records of calls, texts, and data usage, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- **Web-based services**
  - **Social-media:** For keeping records of users, connections between users (such as friend/follows information), posts made by users, rating/like information about posts, etc.
  - **Online retailers:** For keeping records of sales data and orders as for any retailer, but also for tracking a user's product views, search terms, etc., for the purpose of identifying the best items to recommend to that user.
  - **Online advertisements:** For keeping records of click history to enable targeted advertisements, product suggestions, news articles, etc. People access such databases every time they do a web search, make an online purchase, or access a social-networking site.
- **Document databases:** For maintaining collections of new articles, patents, published research papers, etc.
- **Navigation systems:** For maintaining the locations of varies places of interest along with the exact routes of roads, train systems, buses, etc.

Telegram: @uni_k

www.konkur.in

As this list illustrates, databases form an essential part not only of every enterprise but also of a large part of a person's daily activities.

The ways in which people interact with databases has changed over time. Early databases were maintained as back-office systems with which users interacted via printed reports and paper forms for input. As database systems became more sophisticated, better languages were developed for programmers to use in interacting with the data, along with user interfaces that allowed end users within the enterprise to query and update data.

As the support for programmer interaction with databases improved, and computer hardware performance increased even as hardware costs decreased, more sophisticated applications emerged that brought database data into more direct contact not only with end users within an enterprise but also with the general public. Whereas once bank customers had to interact with a teller for every transaction, automated-teller machines (ATMs) allowed direct customer interaction. Today, virtually every enterprise employs web applications or mobile applications to allow its customers to interact directly with the enterprise's database, and, thus, with the enterprise itself.

The user, or customer, can focus on the product or service without being aware of the details of the large database that makes the interaction possible. For instance, when you read a social-media post, or access an online bookstore and browse a book or music collection, you are accessing data stored in a database. When you enter an order online, your order is stored in a database. When you access a bank web site and retrieve your bank balance and transaction information, the information is retrieved from the bank's database system. When you access a web site, information about you may be retrieved from a database to select which advertisements you should see. Almost every interaction with a smartphone results in some sort of database access. Furthermore, data about your web accesses may be stored in a database.

Thus, although user interfaces hide details of access to a database, and most people are not even aware they are dealing with a database, accessing databases forms an essential part of almost everyone's life today.

Broadly speaking, there are two modes in which databases are used.

- The first mode is to support **online transaction processing**, where a large number of users use the database, with each user retrieving relatively small amounts of data, and performing small updates. This is the primary mode of use for the vast majority of users of database applications such as those that we outlined earlier.
- The second mode is to support **data analytics**, that is, the processing of data to draw conclusions, and infer rules or decision procedures, which are then used to drive business decisions.

  For example, banks need to decide whether to give a loan to a loan applicant, online advertisers need to decide which advertisement to show to a particular user. These tasks are addressed in two steps. First, data-analysis techniques attempt to automatically discover rules and patterns from data and create *predictive models*. These models take as input attributes ("features") of individuals, and output pre-

Telegram: @uni_k

www.konkur.in

dictions such as likelihood of paying back a loan, or clicking on an advertisement, which are then used to make the business decision.

As another example, manufacturers and retailers need to make decisions on what items to manufacture or order in what quantities; these decisions are driven significantly by techniques for analyzing past data, and predicting trends. The cost of making wrong decisions can be very high, and organizations are therefore willing to invest a lot of money to gather or purchase required data, and build systems that can use the data to make accurate predictions.

The field of *data mining* combines knowledge-discovery techniques invented by artificial intelligence researchers and statistical analysts with efficient implementation techniques that enable them to be used on extremely large databases.

1.2 Purpose of Database Systems

To understand the purpose of database systems, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating-system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- Add new students, instructors, and courses.
- Register students for courses and generate class rosters.
- Assign grades to students, compute grade point averages (GPA), and generate transcripts.

Programmers develop these application programs to meet the needs of the university.

New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major. As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, and so on. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files.

Keeping organizational information in a file-processing system has a number of major disadvantages:

Telegram: @uni_k

www.konkur.in

- **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures, and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics), the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

- **Difficulty in accessing data**. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* students. The university clerk now has two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of which is satisfactory.

  The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

- **Data isolation**. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

- **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

- **Atomicity problems**. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the

Telegram: @uni_k

www.konkur.in

consistent state that existed prior to the failure. Consider a banking system with a program to transfer $500 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of account *A* but was not credited to the balance of account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

- **Concurrent-access anomalies**. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider account *A*, with a balance of $10,000. If two bank clerks debit the account balance (by say $500 and $100, respectively) of account *A* at almost exactly the same time, the result of the concurrent executions may leave the account balance in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the balance of account *A* may contain either $9500 or $9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

  As another example, suppose a registration program maintains a count of students registered for a course in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

- **Security problems**. Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

Telegram: @uni_k

www.konkur.in

These difficulties, among others, prompted both the initial development of database systems and the transition of file-based applications to database systems, back in the 1960s and 1970s.

In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a university organization as a running example of a typical data-processing application.

1.3 View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

1.3.1 Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

There are a number of different data models that we shall cover in the text. The data models can be classified into four different categories:

- **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model. Chapter 2 and Chapter 7 cover the relational model in detail.
- **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design. Chapter 6 explores it in detail.
- **Semi-structured Data Model**. The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. *JSON* and *Extensible Markup Language* (*XML*) are widely used semi-structured data representations. Semi-structured data models are explored in detail in Chapter 8.

Telegram: @uni_k

www.konkur.in

- **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led initially to the development of a distinct object-oriented data model, but today the concept of objects is well integrated into relational databases. Standards exist to store objects in relational tables. Database systems allow procedures to be stored in the database system and executed by the database system. This can be seen as extending the relational model with notions of encapsulation, methods, and object identity. Object-based data models are summarized in Chapter 8.

A large portion of this text is focused on the relational model because it serves as the foundation for most database applications.

1.3.2 Relational Data Model

In the relational model, data are represented in the form of tables. Each table has multiple columns, and each column has a unique name. Each row of the table represents one piece of information. Figure 1.1 presents a sample relational database comprising two tables: one shows details of university instructors and the other shows details of the various university departments.

The first table, the *instructor* table, shows, for example, that an instructor named Einstein with *ID* 22222 is a member of the Physics department and has an annual salary of $95,000. The second table, *department*, shows, for example, that the Biology department is located in the Watson building and has a budget of $90,000. Of course, a real-world university would have many more departments and instructors. We use small tables in the text to illustrate concepts. A larger example for the same schema is available online.

1.3.3 Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led database system developers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of **data abstraction**, to simplify users' interactions with the system:

- **Physical level**. The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.
- **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data indepen-**

Telegram: @uni_k

www.konkur.in

22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Figure 1.1 A sample relational database.

dence. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

- **View level**. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Figure 1.2 shows the relationship among the three levels of abstraction.

An important feature of data models, such as the relational model, is that they hide such low-level implementation details from not just database users, but even from

Telegram: @uni_k

www.konkur.in

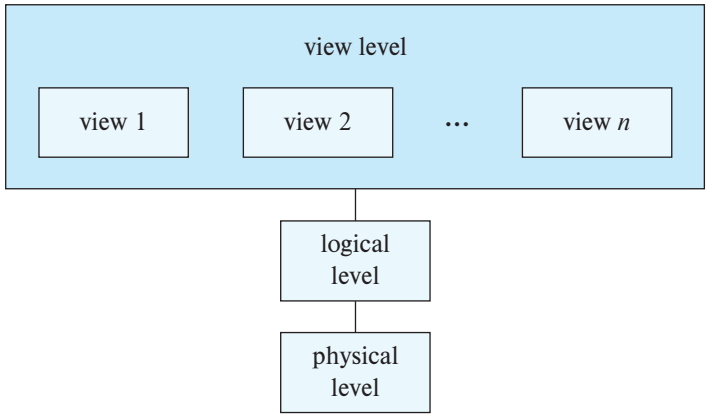


Figure 1.2 The three levels of data abstraction.

database-application developers. The database system allows application developers to store and retrieve data using the abstractions of the data model, and converts the abstract operations into operations on the low-level implementation.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming languages support the notion of a structured type. We may describe the type of a record abstractly as follows:[1]

```
type instructor = record
                    ID : char (5);
                    name : char (20);
                    dept_name : char (20);
                    salary : numeric (8,2);
                  end;
```

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. For example, **char**(20) specifies a string with 20 characters, while **numeric**(8,2) specifies a number with 8 digits, two of which are to the right of the decimal point. A university organization may have several such record types, including:

- *department*, with fields *dept_name*, *building*, and *budget*.
- *course*, with fields *course_id*, *title*, *dept_name*, and *credits*.
- *student*, with fields *ID*, *name*, *dept_name*, and *tot_cred*.

[1]The actual type declaration depends on the language being used. C and C++ use **struct** declarations. Java does not have such a declaration, but a simple class can be defined to the

Telegram: @uni_k

www.konkur.in

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive bytes. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data. For example, there are many possible ways to store tables in files. One way is to store a table as a sequence of records in a file, with a special character (such as a comma) used to delimit the different attributes of a record, and another special character (such as a new-line character) may be used to delimit records. If all attributes have fixed length, the lengths of attributes may be stored separately, and delimiters may be omitted from the file. Variable length attributes could be handled by storing the length, followed by the data. Databases use a type of data structure called an index to support efficient retrieval of records; these too form part of the physical level.

At the logical level, each such record is described by a type definition, as in the previous code segment. The interrelationship of these record types is also defined at the logical level; a requirement that the *dept_name* value of an *instructor* record must appear in the *department* table is an example of such an interrelationship. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

1.3.4 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important in terms of its effect on application programs, since programmers construct

Telegram: @uni_k

www.konkur.in

schema. The physical schema is hidden beneath the logical schema and can usually be changed easily without affecting application programs. Application programs are said to exhibit physical data independence if they do not depend on the physical schema and thus need not be rewritten if the physical schema changes.

We also note that it is possible to create schemas that have problems, such as unnecessarily duplicated information. For example, suppose we store the department *budget* as an attribute of the *instructor* record. Then, whenever the value of the budget for a department (say the Physics department) changes, that change must be reflected in the records of all instructors associated with the department. In Chapter 7, we shall study how to distinguish good schema designs from bad schema designs.

Traditionally, logical schemas were changed infrequently, if at all. Many newer database applications, however, require more flexible logical schemas where, for example, different records in a single relation may have different attributes.

1.4 Database Languages

A database system provides a **data-definition language** (**DDL**) to specify the database schema and a **data-manipulation language** (**DML**) to express database queries and updates. In practice, the data-definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the SQL language. Almost all relational database systems employ the SQL language, which we cover in great detail in Chapter 3, Chapter 4, and Chapter 5.

1.4.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL). The DDL is also used to specify additional properties of the data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain consistency constraints. For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement only those integrity constraints that can be tested with minimal overhead:

- **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it

Telegram: @uni_k

www.konkur.in

can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

- **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists in the university. More precisely, the *dept_name* value in a *course* record must appear in the *dept_name* attribute of some record of the *department* relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The processing of DDL statements, just like those of any other programming language, generates some output. The output of the DDL is placed in the **data dictionary**, which contains **metadata**—that is, data about data. The data dictionary is considered to be a special type of table that can be accessed and updated only by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

1.4.2 The SQL Data-Definition Language

SQL provides a rich DDL that allows one to define tables with data types and integrity constraints.

For instance, the following SQL DDL statement defines the *department* table:

```
create table department
    (dept_name     char (20),
     building      char (15),
     budget        numeric (12,2));
```

Execution of the preceding DDL statement creates the *department* table with three columns: *dept_name*, *building*, and *budget*, each of which has a specific data type associated with it. We discuss data types in more detail in Chapter 3.

The SQL DDL also supports a number of types of integrity constraints. For example, one can specify that the *dept_name* attribute value is a *primary key*, ensuring that no

Telegram: @uni_k

www.konkur.in

two departments can have the same department name. As another example, one can specify that the *dept_name* attribute value appearing in any *instructor* record must also appear in the *dept_name* attribute of some record of the *department* table. We discuss SQL support for integrity constraints and authorizations in Chapter 3 and Chapter 4.

1.4.3 Data-Manipulation Language

A **data-manipulation language** (**DML**) is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database.
- Insertion of new information into the database.
- Deletion of information from the database.
- Modification of information stored in the database.

There are basically two types of data-manipulation language:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
- **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

There are a number of database query languages in use, either commercially or experimentally. We study the most widely used query language, SQL, in Chapter 3 through Chapter 5.

The levels of abstraction that we discussed in Section 1.3 apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system (which we study in Chapter 15 and Chapter 16) translates DML queries into sequences of actions at the physical level of the database system. In Chapter 22, we study the processing of queries in the increasingly common parallel and distributed settings.

Telegram: @uni_k

www.konkur.in

1.4.4 The SQL Data-Manipulation Language

The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table. Here is an example of an SQL query that finds the names of all instructors in the History department:

```
select instructor.name
from instructor
where instructor.dept_name = 'History';
```

The query specifies that those rows from the table *instructor* where the *dept_name* is History must be retrieved, and the *name* attribute of these rows must be displayed. The result of executing this query is a table with a single column labeled *name* and a set of rows, each of which contains the name of an instructor whose *dept_name* is History. If the query is run on the table in Figure 1.1, the result consists of two rows, one with the name El Said and the other with the name Califieri.

Queries may involve information from more than one table. For instance, the following query finds the instructor ID and department name of all instructors associated with a department with a budget of more than $95,000.

```
select instructor.ID, department.dept_name
from instructor, department
where instructor.dept_name= department.dept_name and
      department.budget > 95000;
```

If the preceding query were run on the tables in Figure 1.1, the system would find that there are two departments with a budget of greater than $95,000—Computer Science and Finance; there are five instructors in these departments. Thus, the result consists of a table with two columns (*ID*, *dept_name*) and five rows: (12121, Finance), (45565, Computer Science), (10101, Computer Science), (83821, Computer Science), and (76543, Finance).

1.4.5 Database Access from Application Programs

Non-procedural query languages such as SQL are not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a *host* language, such as C/C++, Java, or Python, with embedded SQL queries that access the data in the database. **Application programs** are programs that are used to interact with the database in this fashion. Examples in a university system are programs that allow students to register for courses, generate class rosters, calculate student GPA, generate payroll checks, and perform other tasks.

Telegram: @uni_k

www.konkur.in

To access the database, DML statements need to be sent from the host to the database where they will be executed. This is most commonly done by using an application-program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve the results. The Open Database Connectivity (ODBC) standard defines application program interfaces for use with C and several other languages. The Java Database Connectivity (JDBC) standard defines a corresponding interface for the Java language.

1.5 Database Design

Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation. They are part of the operation of some enterprise whose end product may be information from the database or may be some device or service for which the database plays only a supporting role.

Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader set of issues. In this text, we focus on the writing of database queries and the design of database schemas, but discuss application design later, in Chapter 9.

A high-level data model provides the database designer with a conceptual framework in which to specify the data requirements of the database users and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.

Next, the designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise. The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. The designer can also examine the design to remove any redundant features. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

In terms of the relational model, the conceptual-design process involves decisions on *what* attributes we want to capture in the database and *how to group* these attributes to form the various tables. The "what" part is basically a business decision, and we shall not discuss it further in this text. The "how" part is mainly a computer-science problem. There are principally two ways to tackle the problem. The first one is to use the entity-relationship model (Chapter 6); the other is to employ a set of algorithms (collectively known as **normalization** that takes as input the set of all attributes and generates a set of tables (Chapter 7).

A fully developed conceptual schema indicates the functional requirements of the enterprise. In a **specification of functional**

Telegram: @uni_k

www.konkur.in

ations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases. In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified. These features include the form of file organization and the internal storage structures; they are discussed in Chapter 13.

1.6 Database Engine

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager, the **query processor** components, and the transaction management component.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases commonly range in size from hundreds of gigabytes to terabytes of data. A gigabyte is approximately 1 billion bytes, or 1000 megabytes (more precisely, 1024 megabytes), while a terabyte is approximately 1 trillion bytes or 1 million megabytes (more precisely, 1024 gigabytes). The largest enterprises have databases that reach into the multi-petabyte range (a petabyte is 1024 terabytes). Since the main memory of computers cannot store this much information, and since the contents of main memory are lost in a system crash, the information is stored on disks. Data are moved between disk storage and main memory as needed. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory. Increasingly, solid-state disks (SSDs) are being used for database storage. SSDs are faster than traditional disks but also more costly.

The query processor is important because it helps the database system to simplify and facilitate access to data. The query processor allows database users to obtain good performance while being able to work at the view level and not be burdened with understanding the physical-level details of the implementation of the system. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

The transaction manager is important because it allows application developers to treat a sequence of database accesses as if they were a single unit that either happens in its entirety or not at all. This permits application developers to think at a higher level of

Telegram: @uni_k

www.konkur.in

abstraction about the application without needing to be concerned with the lower-level details of managing the effects of concurrent access to the data and of system failures.

While database engines were traditionally centralized computer systems, today parallel processing is key for handling very large amounts of data efficiently. Modern database engines pay a lot of attention to parallel data storage and parallel query processing.

1.6.1 Storage Manager

The **storage manager** is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system provided by the operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicts.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which can provide fast access to data items. Like the index in this textbook, a database index provides pointers to those data items that hold a particular value. For example, we could use an index to find the *instructor* record with a particular *ID*, or all *instructor* records with a particular *name*.

Telegram: @uni_k

www.konkur.in

We discuss storage media, file structures, and buffer management in Chapter 12 and Chapter 13. Methods of accessing data efficiently are discussed in Chapter 14.

1.6.2 The Query Processor

The query processor components include:

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query-evaluation engine understands.

  A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**; that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

Query evaluation is covered in Chapter 15, while the methods by which the query optimizer chooses from among the possible evaluation strategies are discussed in Chapter 16.

1.6.3 Transaction Management

Often, several operations on the database form a single logical unit of work. An example is a funds transfer, as in Section 1.2, in which one account A is debited and another account B is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur. That is, the funds transfer must happen in its entirety or not at all. This all-or-none requirement is called **atomicity**. In addition, it is essential that the execution of the funds transfer preserves the consistency of the database. That is, the value of the sum of the balances of A and B must be preserved. This correctness requirement is called **consistency**. Finally, after the successful execution of a funds transfer, the new values of the balances of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called **durability**.

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since

Telegram: @uni_k

www.konkur.in

either the debit of A or the credit of B must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to properly define the various transactions so that each preserves the consistency of the database. For example, the transaction to transfer funds from account A to account B could be defined to be composed of two separate programs: one that debits account A and another that credits account B. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

Ensuring the atomicity and durability properties is the responsibility of the database system itself—specifically, of the **recovery manager**. In the absence of failures, all transactions complete successfully, and atomicity is achieved easily. However, because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform **failure recovery**, that is, it must detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the **concurrency-control manager** to control the interaction among the concurrent transactions, to ensure the consistency of the database. The **transaction manager** consists of the concurrency-control manager and the recovery manager.

The basic concepts of transaction processing are covered in Chapter 17. The management of concurrent transactions is covered in Chapter 18. Chapter 19 covers failure recovery in detail.

The concept of a transaction has been applied broadly in database systems and applications. While the initial use of transactions was in financial applications, the concept is now used in real-time applications in telecommunication, as well as in the management of long-duration activities such as product design or administrative workflows.

1.7 Database and Application Architecture

We are now in a position to provide a single picture of the various components of a database system and the connections among them. Figure 1.3 shows the architecture of a database system that runs on a centralized server machine. The figure summarizes how different types of users interact with a database, and how the different components of a database engine are connected to each other.

The centralized architecture shown in Figure 1.3 is applicable to *shared-memory* server architectures, which have multiple CPUs and exploit parallel processing, but all

Telegram: @uni_k

www.konkur.in

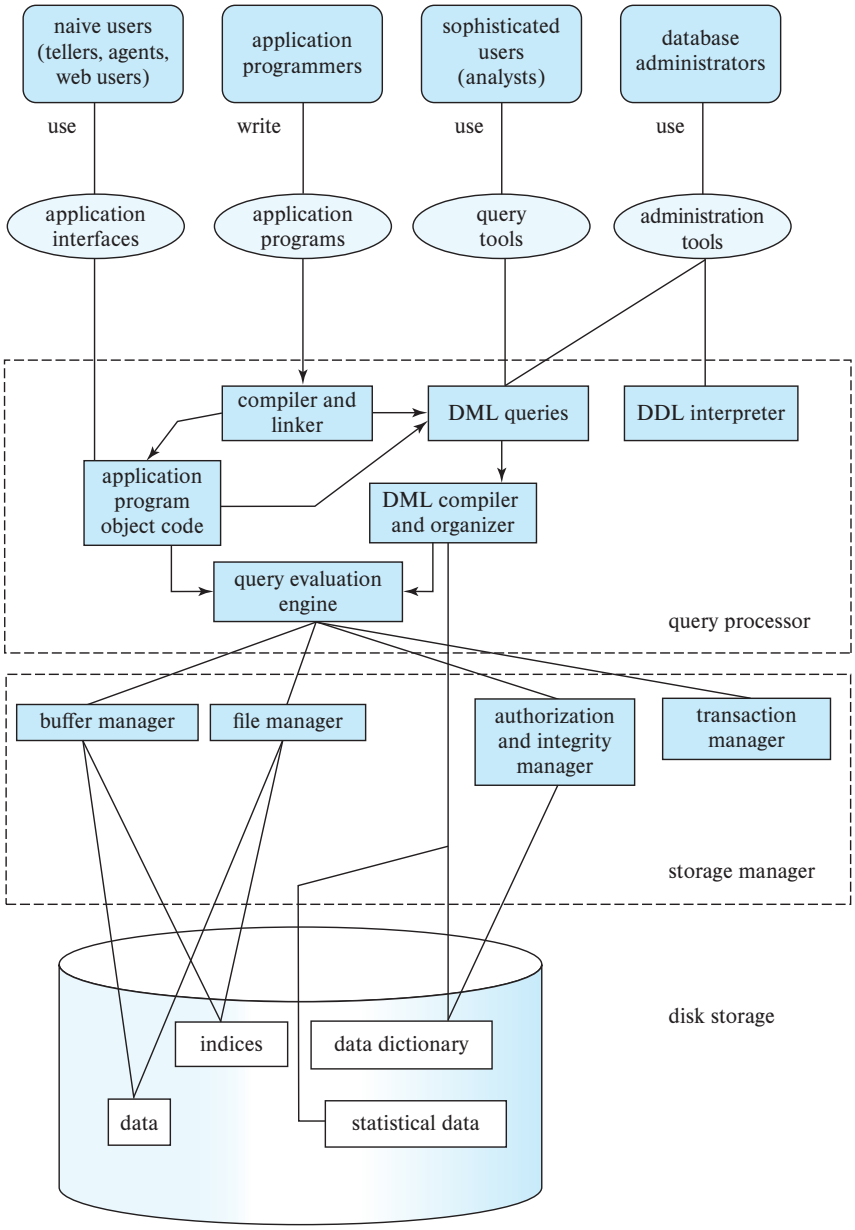


Figure 1.3 System structure.

the CPUs access a common shared memory. To scale up to even larger data volumes and even higher processing speeds, *parallel databases* are designed to run on a cluster consisting of multiple machines. Further, *distributed databases* allow data storage and query processing across multiple geographically separated machines.

Telegram: @uni_k

www.konkur.in

In Chapter 20, we cover the general structure of modern computer systems, with a focus on parallel system architectures. Chapter 21 and Chapter 22 describe how query processing can be implemented to exploit parallel and distributed processing. Chapter 23 presents a number of issues that arise in processing transactions in a parallel or a distributed database and describes how to deal with each issue. The issues include how to store data, how to ensure atomicity of transactions that execute at multiple sites, how to perform concurrency control, and how to provide high availability in the presence of failures.

We now consider the architecture of applications that use databases as their back-end. Database applications can be partitioned into two or three parts, as shown in Figure 1.4. Earlier-generation database applications used a **two-tier architecture**, where the application resides at the client machine, and invokes database system functionality at the server machine through query language statements.

In contrast, modern database applications use a **three-tier architecture**, where the client machine acts as merely a front end and does not contain any direct database calls; web browsers and mobile applications are the most commonly used application clients today. The front end communicates with an **application server**. The application server, in turn, communicates with a database system to access data. The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications provide better security as well as better performance than two-tier applications.

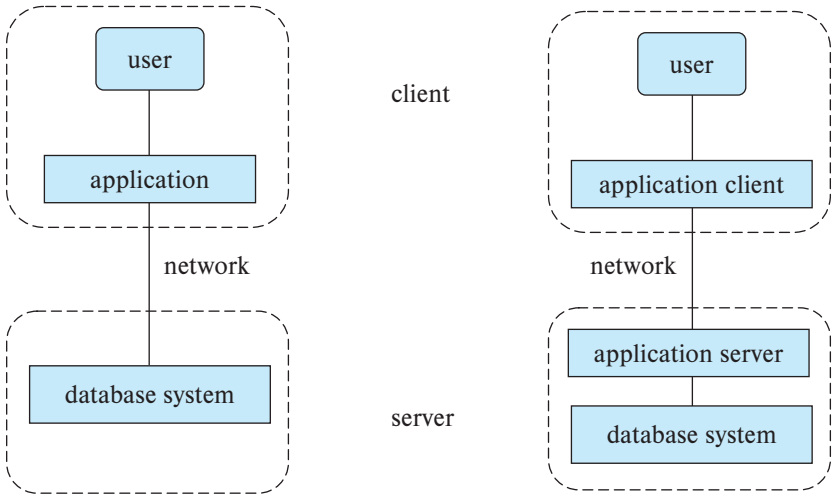


Figure 1.4 Two-tier and three-tier architectures.

Telegram: @uni_k

www.konkur.in

1.8 Database Users and Administrators

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

1.8.1 Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

- **Naïve users** are unsophisticated users who interact with the system by using predefined user interfaces, such as web or mobile applications. The typical user interface for naïve users is a forms interface, where the user can fill in appropriate fields of the form. Naïve users may also view read *reports* generated from the database.

  As an example, consider a student, who during class registration period, wishes to register for a class by using a web interface. Such a user connects to a web application program that runs at a web server. The application first verifies the identity of the user and then allows her to access a form where she enters the desired information. The form information is sent back to the web application at the server, which then determines if there is room in the class (by retrieving information from the database) and if so adds the student information to the class roster in the database.

- **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.

- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.

1.8.2 Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**). The functions of a DBA include:

- **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

- **Storage structure and access-method definition**. The DBA may specify some parameters pertaining to the physical organization of the data and the indices to be created.

Telegram: @uni_k

www.konkur.in

- **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever a user tries to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
  - Periodically backing up the database onto remote servers, to prevent loss of data in case of disasters such as flooding.
  - Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
  - Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.9 History of Database Systems

Information processing drives the growth of computers, as it has from the earliest days of commercial computers. In fact, automation of data processing tasks predates computers. Punched cards, invented by Herman Hollerith, were used at the very beginning of the twentieth century to record U.S. census data, and mechanical systems were used to process the cards and tabulate results. Punched cards were later widely used as a means of entering data into computers.

Techniques for data storage and processing have evolved over the years:

- **1950s and early 1960s**: Magnetic tapes were developed for data storage. Data-processing tasks such as payroll were automated, with data stored on tapes. Processing of data consisted of reading data from one or more tapes and writing data to a new tape. Data could also be input from punched card decks and output to printers. For example, salary raises were processed by entering the raises on punched cards and reading the punched card deck in synchronization with a tape containing the master salary details. The records had to be in the same sorted order. The salary raises would be added to the salary read from the master tape and written to a new tape; the new tape would become the new master tape.

  Tapes (and card decks) could be read only sequentially, and data sizes were much larger than main memory; thus, data-processing programs were forced to

Telegram: @uni_k

www.konkur.in

process data in a particular order by reading and merging data from tapes and card decks.

- **Late 1960s and early 1970s**: Widespread use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data. The position of data on disk was immaterial, since any location on disk could be accessed in just tens of milliseconds. Data were thus freed from the tyranny of sequentiality. With the advent of disks, the network and hierarchical data models were developed, which allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.

  A landmark paper by Edgar Codd in 1970 defined the relational model and nonprocedural ways of querying data in the relational model, and relational databases were born. The simplicity of the relational model and the possibility of hiding implementation details completely from the programmer were enticing indeed. Codd later won the prestigious Association of Computing Machinery Turing Award for his work.

- **Late 1970s and 1980s**: Although academically interesting, the relational model was not used in practice initially because of its perceived performance disadvantages; relational databases could not match the performance of existing network and hierarchical databases. That changed with System R, a groundbreaking project at IBM Research that developed techniques for the construction of an efficient relational database system. The fully functional System R prototype led to IBM's first relational database product, SQL/DS. At the same time, the Ingres system was being developed at the University of California at Berkeley. It led to a commercial product of the same name. Also around this time, the first version of Oracle was released. Initial commercial relational database systems, such as IBM DB2, Oracle, Ingres, and DEC Rdb, played a major role in advancing techniques for efficient processing of declarative queries.

  By the early 1980s, relational databases had become competitive with network and hierarchical database systems even in the area of performance. Relational databases were so easy to use that they eventually replaced network and hierarchical databases. Programmers using those older models were forced to deal with many low-level implementation details, and they had to code their queries in a procedural fashion. Most importantly, they had to keep efficiency in mind when designing their programs, which involved a lot of effort. In contrast, in a relational database, almost all these low-level tasks are carried out automatically by the database system, leaving the programmer free to work at a logical level. Since attaining dominance in the 1980s, the relational model has reigned supreme among data models.

  The 1980s also saw much research on parallel and distributed databases, as well as initial work on object-oriented databases.

Telegram: @uni_k

www.konkur.in

- **1990s**: The SQL language was designed primarily for decision support applications, which are query-intensive, yet the mainstay of databases in the 1980s was transaction-processing applications, which are update-intensive.

  In the early 1990s, decision support and querying re-emerged as a major application area for databases. Tools for analyzing large amounts of data saw a large growth in usage. Many database vendors introduced parallel database products in this period. Database vendors also began to add object-relational support to their databases.

  The major event of the 1990s was the explosive growth of the World Wide Web. Databases were deployed much more extensively than ever before. Database systems now had to support very high transaction-processing rates, as well as very high reliability and 24 × 7 availability (availability 24 hours a day, 7 days a week, meaning no downtime for scheduled maintenance activities). Database systems also had to support web interfaces to data.

- **2000s**: The types of data stored in database systems evolved rapidly during this period. Semi-structured data became increasingly important. XML emerged as a data-exchange standard. JSON, a more compact data-exchange format well suited for storing objects from JavaScript or other programming languages subsequently grew increasingly important. Increasingly, such data were stored in relational database systems as support for the XML and JSON formats was added to the major commercial systems. Spatial data (that is, data that include geographic information) saw widespread use in navigation systems and advanced applications. Database systems added support for such data.

  Open-source database systems, notably PostgreSQL and MySQL saw increased use. "Auto-admin" features were added to database systems in order to allow automatic reconfiguration to adapt to changing workloads. This helped reduce the human workload in administering a database.

  Social network platforms grew at a rapid pace, creating a need to manage data about connections between people and their posted data, that did not fit well into a tabular row-and-column format. This led to the development of graph databases.

  In the latter part of the decade, the use of data analytics and **data mining** in enterprises became ubiquitous. Database systems were developed specifically to serve this market. These systems featured physical data organizations suitable for analytic processing, such as "column-stores," in which tables are stored by column rather than the traditional row-oriented storage of the major commercial database systems.

  The huge volumes of data, as well as the fact that much of the data used for analytics was textual or semi-structured, led to the development of programming frameworks, such as *map-reduce*, to facilitate application programmers' use of parallelism in analyzing data. In time, support for these features migrated into traditional database systems. Even in the late 2010s, debate continued in the database

Telegram: @uni_k

www.konkur.in

research community over the relative merits of a single database system serving both traditional transaction processing applications and the newer data-analysis applications versus maintaining separate systems for these roles.

The variety of new data-intensive applications and the need for rapid development, particularly by startup firms, led to "NoSQL" systems that provide a lightweight form of data management. The name was derived from those systems' lack of support for the ubiquitous database query language SQL, though the name is now often viewed as meaning "not only SQL." The lack of a high-level query language based on the relational model gave programmers greater flexibility to work with new types of data. The lack of traditional database systems' support for strict data consistency provided more flexibility in an application's use of distributed data stores. The NoSQL model of "eventual consistency" allowed for distributed copies of data to be inconsistent as long they would eventually converge in the absence of further updates.

- **2010s**: The limitations of NoSQL systems, such as lack of support for consistency, and lack of support for declarative querying, were found acceptable by many applications (e.g., social networks), in return for the benefits they provided such as scalability and availability. However, by the early 2010s it was clear that the limitations made life significantly more complicated for programmers and database administrators. As a result, these systems evolved to provide features to support stricter notions of consistency, while continuing to support high scalability and availability. Additionally, these systems increasingly support higher levels of abstraction to avoid the need for programmers to have to reimplement features that are standard in a traditional database system.

  Enterprises are increasingly outsourcing the storage and management of their data. Rather than maintaining in-house systems and expertise, enterprises may store their data in "cloud" services that host data for various clients in multiple, widely distributed server farms. Data are delivered to users via web-based services. Other enterprises are outsourcing not only the storage of their data but also whole applications. In such cases, termed "software as a service," the vendor not only stores the data for an enterprise but also runs (and maintains) the application software. These trends result in significant savings in costs, but they create new issues not only in responsibility for security breaches, but also in data ownership, particularly in cases where a government requests access to data.

  The huge influence of data and data analytics in daily life has made the management of data a frequent aspect of the news. There is an unresolved tradeoff between an individual's right of privacy and society's need to know. Various national governments have put regulations on privacy in place. High-profile security breaches have created a public awareness of the challenges in cybersecurity and the risks of storing data.

Telegram: @uni_k

www.konkur.in

1.10 Summary

- A database-management system (DBMS) consists of a collection of interrelated data and a collection of programs to access those data. The data describe one particular enterprise.
- The primary goal of a DBMS is to provide an environment that is both convenient and efficient for people to use in retrieving and storing information.
- Database systems are ubiquitous today, and most people interact, either directly or indirectly, with databases many times every day.
- Database systems are designed to store large bodies of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information. In addition, the database system must provide for the safety of the information stored in the face of system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.
- A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.
- Underlying the structure of a database is the data model: a collection of conceptual tools for describing data, data relationships, data semantics, and data constraints.
- The relational data model is the most widely deployed model for storing data in databases. Other data models are the object-oriented model, the object-relational model, and semi-structured data models.
- A data-manipulation language (DML) is a language that enables users to access or manipulate data. Nonprocedural DMLs, which require a user to specify only what data are needed, without specifying exactly how to get those data, are widely used today.
- A data-definition language (DDL) is a language for specifying the database schema and other properties of the data.
- Database design mainly involves the design of the database schema. The entity-relationship (E-R) data model is a widely used model for database design. It provides a convenient graphical representation to view data, relationships, and constraints.
- A database system has several subsystems.
    - The storage manager subsystem provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

Telegram: @uni_k

www.konkur.in

  - The query processor subsystem compiles and executes DDL and DML statements.
- Transaction management ensures that the database remains in a consistent (correct) state despite system failures. The transaction manager ensures that concurrent transaction executions proceed without conflicts.
- The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or parallel, involving multiple machines. Distributed databases span multiple geographically separated machines.
- Database applications are typically broken up into a front-end part that runs at client machines and a part that runs at the backend. In two-tier architectures, the front end directly communicates with a database running at the back end. In three-tier architectures, the back end part is itself broken up into an application server and a database server.
- There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.
- Data-analysis techniques attempt to automatically discover rules and patterns from data. The field of data mining combines knowledge-discovery techniques invented by artificial intelligence researchers and statistical analysts with efficient implementation techniques that enable them to be used on extremely large databases.

Review Terms

- Database-management system (DBMS)
- Database-system applications
- Online transaction processing
- Data analytics
- File-processing systems
- Data inconsistency
- Consistency constraints
- Data abstraction
  - Physical level
  - Logical level
  - View level
- Instance
- Schema
  - Physical schema
  - Logical schema
  - Subschema
- Physical data independence
- Data models
  - Entity-relationship model
  - Relational data model
  - Semi-structured data model
  - Object-based data model

Telegram: @uni_k

www.konkur.in

- Database languages
  - Data-definition language
  - Data-manipulation language
    - Procedural DML
    - Declarative DML
    - nonprocedural DML
  - Query language
- Data-definition language
  - Domain Constraints
  - Referential Integrity
  - Authorization
    - Read authorization
    - Insert authorization
    - Update authorization
    - Delete authorization
- Metadata
- Application program
- Database design
  - Conceptual design
  - Normalization
  - Specification of functional requirements
  - Physical-design phase
- Database Engine
  - Storage manager
    - Authorization and integrity manager
    - Transaction manager
    - File manager
    - Buffer manager
    - Data files
    - Data dictionary
    - Indices
  - Query processor
    - DDL interpreter
    - DML compiler
    - Query optimization
    - Query evaluation engine
  - Transactions
    - Atomicity
    - Consistency
    - Durability
    - Recovery manager
    - Failure recovery
    - Concurrency-control manager
- Database Architecture
  - Centralized
  - Parallel
  - Distributed
- Database Application Architecture
  - Two-tier
  - Three-tier
  - Application server
- Database administrator (DBA)

Practice Exercises

1.1 This chapter has described several major advantages of a database system. What are two disadvantages?

1.2 List five ways in which the type declaration system of a language such as Java or C++ differs from the data definition language used in a database.

Telegram: @uni_k

www.konkur.in

1.3 List six major steps that you would take in setting up a database for a particular enterprise.

1.4 Suppose you want to build a video site similar to YouTube. Consider each of the points listed in Section 1.2 as disadvantages of keeping data in a file-processing system. Discuss the relevance of each of these points to the storage of actual video data, and to metadata about the video, such as title, the user who uploaded it, tags, and which users viewed it.

1.5 Keyword queries used in web search are quite different from database queries. List key differences between the two, in terms of the way the queries are specified and in terms of what is the result of a query.

Exercises

1.6 List four applications you have used that most likely employed a database system to store persistent data.

1.7 List four significant differences between a file-processing system and a DBMS.

1.8 Explain the concept of physical data independence and its importance in database systems.

1.9 List five responsibilities of a database-management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.

1.10 List at least two reasons why database systems support data manipulation using a declarative query language such as SQL, instead of just providing a library of C or C++ functions to carry out data manipulation.

1.11 Assume that two students are trying to register for a course in which there is only one open seat. What component of a database system prevents both students from being given that last seat?

1.12 Explain the difference between two-tier and three-tier application architectures. Which is better suited for web applications? Why?

1.13 List two features developed in the 2000s and that help database systems handle data-analytics workloads.

1.14 Explain why NoSQL systems emerged in the 2000s, and briefly contrast their features with traditional database systems.

1.15 Describe at least three tables that might be used to store information in a social-networking system such as Facebook.

Telegram: @uni_k

www.konkur.in

Tools

There are a large number of commercial database systems in use today. The major ones include: IBM DB2 (www.ibm.com/software/data/db2), Oracle (www.oracle.com), Microsoft SQL Server (www.microsoft.com/sql), IBM Informix (www.ibm.com/software/data/informix), SAP Adaptive Server Enterprise (formerly Sybase) (www.sap.com/products/sybase-ase.html), and SAP HANA (www.sap.com/products/hana.html). Some of these systems are available free for personal or non-commercial use, or for development, but are not free for actual deployment.

There are also a number of free/public domain database systems; widely used ones include MySQL (www.mysql.com), PostgreSQL (www.postgresql.org), and the embedded database SQLite (www.sqlite.org).

A more complete list of links to vendor web sites and other information is available from the home page of this book, at db-book.com.

Further Reading

[Codd (1970)] is the landmark paper that introduced the relational model. Textbook coverage of database systems is provided by [O'Neil and O'Neil (2000)], [Ramakrishnan and Gehrke (2002)], [Date (2003)], [Kifer et al. (2005)], [Garcia-Molina et al. (2008)], and [Elmasri and Navathe (2016)], in addition to this textbook,

A review of accomplishments in database management and an assessment of future research challenges appears in [Abadi et al. (2016)]. The home page of the ACM Special Interest Group on Management of Data (www.acm.org/sigmod) provides a wealth of information about database research. Database vendor web sites (see the Tools section above) provide details about their respective products.

Bibliography

[Abadi et al. (2016)] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. RÂ´e, D. Suciu, M. Stonebraker, T. Walter, and J. Widom, "The Beckman Report on Database Research", *Communications of the ACM*, Volume 59, Number 2 (2016), pages 92–99.

[Codd (1970)] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Date (2003)] C. J. Date, *An Introduction to Database Systems*, 8th edition, Addison Wesley (2003).

Telegram: @uni_k

www.konkur.in

[Elmasri and Navathe (2016)] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th edition, Addison Wesley (2016).

[Garcia-Molina et al. (2008)] H. Garcia-Molina, J. D. Ullman, and J. D. Widom, *Database Systems: The Complete Book*, 2nd edition, Prentice Hall (2008).

[Kifer et al. (2005)] M. Kifer, A. Bernstein, and P. Lewis, *Database Systems: An Application Oriented Approach, Complete Version*, 2nd edition, Addison Wesley (2005).

[O'Neil and O'Neil (2000)] P. O'Neil and E. O'Neil, *Database: Principles, Programming, Performance*, 2nd edition, Morgan Kaufmann (2000).

[Ramakrishnan and Gehrke (2002)] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd edition, McGraw Hill (2002).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in



PART 1

RELATIONAL LANGUAGES

A data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. The relational model uses a collection of tables to represent both data and the relationships among those data. Its conceptual simplicity has led to its widespread adoption; today a vast majority of database products are based on the relational model. The relational model describes data at the logical and view levels, abstracting away low-level details of data storage.

To make data from a relational database available to users, we have to address how users specify requests for retrieving and updating data. Several query languages have been developed for this task, which are covered in this part.

Chapter 2 introduces the basic concepts underlying relational databases, including the coverage of relational algebra—a formal query language that forms the basis for SQL. The language SQL is the most widely used relational query language today and is covered in great detail in this part.

Chapter 3 provides an overview of the SQL query language, including the SQL data definition, the basic structure of SQL queries, set operations, aggregate functions, nested subqueries, and modification of the database.

Chapter 4 provides further details of SQL, including join expressions, views, transactions, integrity constraints that are enforced by the database, and authorization mechanisms that control what access and update actions can be carried out by a user.

Chapter 5 covers advanced topics related to SQL including access to SQL from programming languages, functions, procedures, triggers, recursive queries, and advanced aggregation features.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



CHAPTER 2

Introduction to the Relational Model

The relational model remains the primary data model for commercial data-processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. It has retained this position by incorporating various new features and capabilities over its half-century of existence. Among those additions are object-relational features such as complex data types and stored procedures, support for XML data, and various tools to support semi-structured data. The relational model's independence from any specific underlying low-level data structures has allowed it to persist despite the advent of new approaches to data storage, including modern column-stores that are designed for large-scale data mining.

In this chapter, we first study the fundamentals of the relational model. A substantial theory exists for relational databases. In Chapter 6 and Chapter 7, we shall examine aspects of database theory that help in the design of relational database schemas, while in Chapter 15 and Chapter 16 we discuss aspects of the theory dealing with efficient processing of queries. In Chapter 27, we study aspects of formal relational languages beyond our basic coverage in this chapter.

2.1 Structure of Relational Databases

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure 2.1, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept_name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept_name*, and *salary*. Similarly, the *course* table of Figure 2.2 stores information about courses, consisting of a *course_id*, *title*, *dept_name*, and *credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course id*.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.1 The *instructor* relation.

Figure 2.3 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course_id* and *prereq_id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other. As another example, when we consider the table *instructor*, a row in the table can be thought of as representing the relationship

BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.2 The *course* relation.

Telegram: @uni_k

www.konkur.in

BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 2.3 The *prereq* relation.

between a specified *ID* and the corresponding values for *name*, *dept_name*, and *salary* values.

In general, a row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between n values is represented mathematically by an *n-tuple* of values, that is, a tuple with n values, which corresponds to a row in a table.

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure 2.1, we can see that the relation *instructor* has four attributes: *ID*, *name*, *dept_name*, and *salary*.

We use the term **relation instance** to refer to a specific instance of a relation, that is, containing a specific set of rows. The instance of *instructor* shown in Figure 2.1 has 12 tuples, corresponding to 12 instructors.

In this chapter, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. To simplify our presentation, we exclude much of the data an actual university database would contain. We shall discuss criteria for the appropriateness of relational structures in great detail in Chapter 6 and Chapter 7.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 2.1, or are unsorted, as in Figure 2.4, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we generally show the relations sorted by their first attribute.

For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

Telegram: @uni_k

www.konkur.in

22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Figure 2.4 Unsorted display of the *instructor* relation.

We require that, for all relations r, the domains of all attributes of r be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, suppose the table *instructor* had an attribute *phone_number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone_number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely, the individual phone numbers in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone_number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code, and a local number, we would be treating it as a non-atomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone_number* would have an atomic domain.

The **null value** is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone_number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus they should be eliminated if at all possible. We shall assume null values are absent initially, and in Section 3.6 we describe the effect of nulls on different operations.

The relatively strict structure of relations results in several important practical advantages in the storage and processing of data, as we shall see. That strict structure is suitable for well-defined and relatively static applications, but it is less suitable for applications where not only data but also the types and structure of those data change over time. A modern enterprise needs to find a good balance between the efficiencies of structured data and those situations where a predetermined structure is limiting.

Telegram: @uni_k

www.konkur.in

2.2 Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time.

The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. We shall not be concerned about the precise definition of the domain of each attribute until we discuss the SQL language in Chapter 3.

The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change.

Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *instructor*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example "the *instructor* schema," or "an instance of the *instructor* relation." However, where it is clear whether we mean the schema or the instance, we simply use the relation name.

Consider the *department* relation of Figure 2.5. The schema for that relation is:

department (*dept_name*, *building*, *budget*)

Note that the attribute *dept_name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations. For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the *department* relation to find the *dept_name* of all the departments housed in Watson. Then, for each such department, we look in

Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.

Telegram: @uni_k

www.konkur.in

BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

Figure 2.6 The *section* relation.

the *instructor* relation to find the information about the instructor associated with the corresponding *dept_name*.

Each course in a university may be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individual offering, or section, of the class. The schema is:

section (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)

Figure 2.6 shows a sample instance of the *section* relation.

We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is:

teaches (*ID*, *course_id*, *sec_id*, *semester*, *year*)

Figure 2.7 shows a sample instance of the *teaches* relation.

As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *instructor*, *department*, *course*, *section*, *prereq*, and *teaches*, we use the following relations in this text:

- *student* (*ID*, *name*, *dept_name*, *tot_cred*)
- *advisor* (*s_id*, *i_id*)
- *takes* (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)

Telegram: @uni_k

www.konkur.in

10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure 2.7 The *teaches* relation.

- *classroom* (*building*, *room_number*, *capacity*)
- *time_slot* (*time_slot_id*, *day*, *start_time*, *end_time*)

2.3 Keys

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.[1]

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name.

Formally, let R denote the set of attributes in the schema of relation r. If we say that a subset K of R is a *superkey* for r, we are restricting consideration to instances of relations r in which no two distinct tuples have the same values on all attributes in K. That is, if t_1 and t_2 are in r and $t_1 \neq t_2$, then $t_1.K \neq t_2.K$.

[1] Commercial database systems relax the requirement that a relation is a set and instead allow duplicate tuples. This is discussed further in Chapter 3.

Telegram: @uni_k

www.konkur.in

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If K is a superkey, then so is any superset of K. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept_name* is sufficient to distinguish among members of the *instructor* relation. Then, both {*ID*} and {*name*, *dept_name*} are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, {*ID*, *name*}, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled. Thus, primary keys are also referred to as **primary key constraints**.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

Consider the *classroom* relation:

classroom (*building*, *room_number*, *capacity*)

Here the primary key consists of two attributes, *building* and *room_number*, which are underlined to indicate they are part of the primary key. Neither attribute by itself can uniquely identify a classroom, although together they uniquely identify a classroom. Also consider the *time_slot* relation:

time_slot (*time_slot_id*, *day*, *start_time*, *end_time*)

Each section has an associated *time_slot_id*. The *time_slot* relation provides information on which days of the week, and at what times, a particular *time_slot_id* meets. For example, *time_slot_id* 'A' may meet from 8.00 AM to 8.50 AM on Mondays, Wednesdays, and Fridays. It is possible for a time slot to have multiple sessions within a single day, at different times, so the *time_slot_id* and *day* together do not uniquely identify the tuple. The primary key of the *time_slot* relation thus consists of the attributes *time_slot_id*, *day*, and *start_time*, since these three attributes together uniquely identify a time slot for a course.

Primary keys must be chosen with care. As we noted, the name of a person is insufficient, because there may be many people with the same name. In the United States, the social security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social security numbers, international enterprises must

Telegram: @uni_k

www.konkur.in

generate their own unique identifiers. An alternative is to use some unique combination of other attributes as a key.

The primary key should be chosen such that its attribute values are never, or are very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

Figure 2.8 shows the complete set of relations that we use in our sample university schema, with primary-key attributes underlined.

Next, we consider another type of constraint on the contents of relations, called foreign-key constraints. Consider the attribute *dept_name* of the *instructor* relation. It would not make sense for a tuple in *instructor* to have a value for *dept_name* that does not correspond to a department in the *department* relation. Thus, in any database instance, given any tuple, say t_a, from the *instructor* relation, there must be some tuple, say t_b, in the *department* relation such that the value of the *dept_name* attribute of t_a is the same as the value of the primary key, *dept_name*, of t_b.

A **foreign-key constraint** from attribute(s) A of relation r_1 to the primary-key B of relation r_2 states that on any database instance, the value of A for each tuple in r_1 must also be the value of B for some tuple in r_2. Attribute set A is called a **foreign key** from r_1, referencing r_2. The relation r_1 is also called the **referencing relation** of the foreign-key constraint, and r_2 is called the **referenced relation**.

For example, the attribute *dept_name* in *instructor* is a foreign key from *instructor*, referencing *department*; note that *dept_name* is the primary key of *department*. Similarly,

---

classroom(<u>*building*</u>, <u>*room_number*</u>, *capacity*)
department(<u>*dept_name*</u>, *building*, *budget*)
course(<u>*course_id*</u>, *title*, *dept_name*, *credits*)
instructor(<u>*ID*</u>, *name*, *dept_name*, *salary*)
section(<u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *building*, *room_number*, *time_slot_id*)
teaches(<u>*ID*</u>, <u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>)
student(<u>*ID*</u>, *name*, *dept_name*, *tot_cred*)
takes(<u>*ID*</u>, <u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *grade*)
advisor(<u>*s_ID*</u>, *i_ID*)
time_slot(<u>*time_slot_id*</u>, <u>*day*</u>, <u>*start_time*</u>, *end_time*)
prereq(<u>*course_id*</u>, <u>*prereq_id*</u>)

Figure 2.8 Schema of the university database.

Telegram: @uni_k

www.konkur.in

the attributes *building* and *room_number* of the *section* relation together form a foreign key referencing the *classroom* relation.

Note that in a foreign-key constraint, the referenced attribute(s) must be the primary key of the referenced relation. The more general case, a referential-integrity constraint, relaxes the requirement that the referenced attributes form the primary key of the referenced relation.

As an example, consider the values in the *time_slot_id* attribute of the *section* relation. We require that these values must exist in the *time_slot_id* attribute of the *time_slot* relation. Such a requirement is an example of a referential integrity constraint. In general, a **referential integrity constraint** requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

Note that *time_slot* does not form a primary key of the *time_slot* relation, although it is a part of the primary key; thus, we cannot use a foreign-key constraint to enforce the above constraint. In fact, foreign-key constraints are a *special case* of referential integrity constraints, where the referenced attributes form the primary key of the referenced relation. Database systems today typically support foreign-key constraints, but they do not support referential integrity constraints where the referenced attribute is not a primary key.

2.4 Schema Diagrams

A database schema, along with primary key and foreign-key constraints, can be depicted by **schema diagrams**. Figure 2.9 shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue and the attributes listed inside the box.

Primary-key attributes are shown underlined. Foreign-key constraints appear as arrows from the foreign-key attributes of the referencing relation to the primary key of the referenced relation. We use a two-headed arrow, instead of a single-headed arrow, to indicate a referential integrity constraint that is not a foreign-key constraints. In Figure 2.9, the line with a two-headed arrow from *time_slot_id* in the *section* relation to *time_slot_id* in the *time_slot* relation represents the referential integrity constraint from *section.time_slot_id* to *time_slot.time_slot_id*.

Many database systems provide design tools with a graphical user interface for creating schema diagrams.[2] We shall discuss a different diagrammatic representation of schemas, called the entity-relationship diagram, at length in Chapter 6; although there are some similarities in appearance, these two notations are quite different, and should not be confused for one another.

[2]The two-headed arrow notation to represent referential integrity constraints has been introduced by us and is not supported by any tool as far as we know; the notations for primary and

Telegram: @uni_k

www.konkur.in

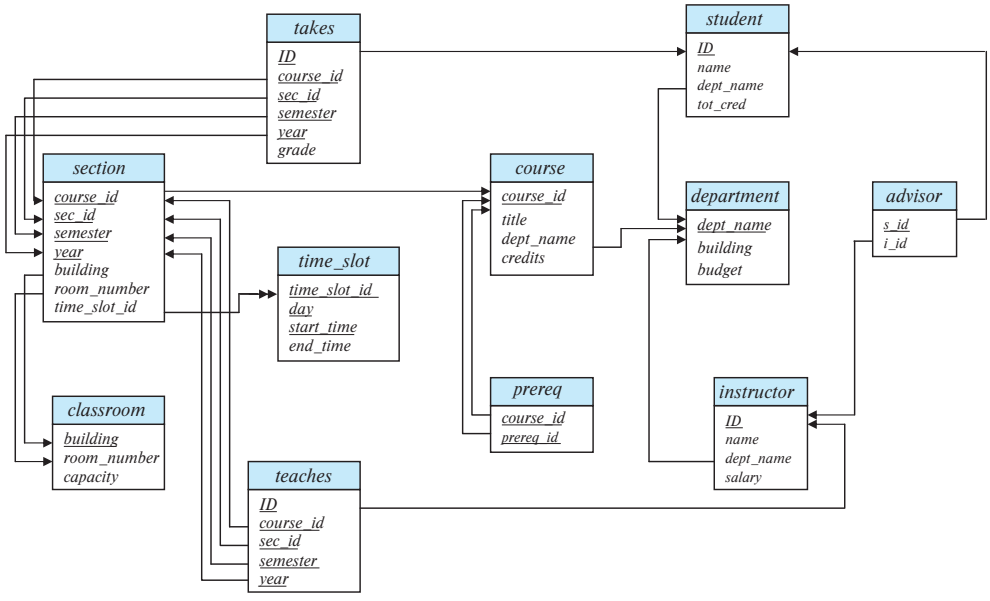


Figure 2.9 Schema diagram for the university database.

2.5 Relational Query Languages

A **query language** is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language. Query languages can be categorized as imperative, functional, or declarative. In an **imperative query language**, the user instructs the system to perform a specific sequence of operations on the database to compute the desired result; such languages usually have a notion of state variables, which are updated in the course of the computation.

In a **functional query language**, the computation is expressed as the evaluation of functions that may operate on data in the database or on the results of other functions; functions are side-effect free, and they do not update the program state.[3] In a **declarative query language**, the user describes the desired information without giving a specific sequence of steps or function calls for obtaining that information; the desired information is typically described using some form of mathematical logic. It is the job of the database system to figure out how to obtain the desired information.

[3]The term *procedural language* has been used in earlier editions of the book to refer to languages based on procedure invocations, which include functional languages; however, the term is also widely used to refer to imperative languages. To avoid confusion we no longer use the term.

Telegram: @uni_k

www.konkur.in

There are a number of "pure" query languages.

- The *relational algebra*, which we describe in Section 2.6, is a functional query language.[4] The relational algebra forms the theoretical basis of the SQL query language.
- The tuple relational calculus and domain relational calculus, which we describe in Chapter 27 (available online) are declarative.

These query languages are terse and formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

Query languages used in practice, such as the SQL query language, include elements of the imperative, functional, and declarative approaches. We study the very widely used query language SQL in Chapter 3 through Chapter 5.

2.6 The Relational Algebra

The relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result.

Some of these operations, such as the select, project, and rename operations, are called *unary* operations because they operate on one relation. The other operations, such as union, Cartesian product, and set difference, operate on pairs of relations and are, therefore, called *binary* operations.

Although the relational algebra operations form the basis for the widely used SQL query language, database systems do not allow users to write queries in relational algebra. However, there are implementations of relational algebra that have been built for students to practice relational algebra queries. The website of our book, db-book.com, under the link titled Laboratory Material, provides pointers to a few such implementations.

It is worth recalling at this point that since a relation is a set of tuples, relations cannot contain duplicate tuples. In practice, however, tables in database systems are permitted to contain duplicates unless a specific constraint prohibits it. But, in discussing the formal relational algebra, we require that duplicates be eliminated, as is required by the mathematical definition of a set. In Chapter 3 we discuss how relational algebra can be extended to work on multisets, which are sets that can contain duplicates.

[4] Unlike modern functional languages, relational algebra supports only a small number of predefined functions, which define an algebra on relations.

Telegram: @uni_k

www.konkur.in

22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Figure 2.10 Result of $\sigma_{dept_name = \text{"Physics"}}(instructor)$.

2.6.1 The Select Operation

The **select** operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the *instructor* relation where the instructor is in the "Physics" department, we write:

$$\sigma_{dept_name = \text{"Physics"}}(instructor)$$

If the *instructor* relation is as shown in Figure 2.1, then the relation that results from the preceding query is as shown in Figure 2.10.

We can find all instructors with salary greater than $90,000 by writing:

$$\sigma_{salary > 90000}(instructor)$$

In general, we allow comparisons using $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Thus, to find the instructors in Physics with a salary greater than $90,000, we write:

$$\sigma_{dept_name = \text{"Physics"} \wedge salary > 90000}(instructor)$$

The selection predicate may include comparisons between two attributes. To illustrate, consider the relation *department*. To find all departments whose name is the same as their building name, we can write:

$$\sigma_{dept_name = building}(department)$$

2.6.2 The Project Operation

Suppose we want to list all instructors' *ID*, *name*, and *salary*, but we do not care about the *dept_name*. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. We write the query to produce such a list as:

$$\Pi_{ID,\ name,\ salary}(instructor)$$

Figure 2.11 shows the relation that results from this query.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Figure 2.11 Result of $\Pi_{ID,\ name,\ salary}(instructor)$.

The basic version of the project operator $\Pi_L(E)$ allows only attribute names to be present in the list L. A generalized version of the operator allows expressions involving attributes to appear in the list L. For example, we could use:

$$\Pi_{ID,name,salary/12}(instructor)$$

to get the monthly salary of each instructor.

2.6.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find the names of all instructors in the Physics department." We write:

$$\Pi_{name}\ (\sigma_{dept_name = \text{"Physics"}}\ (instructor))$$

Notice that, instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a **relational-algebra expression**. Composing relational-algebra operations into relational-algebra expressions is just like composing arithmetic operations (such as $+$, $-$, $*$, and $\div$) into arithmetic expressions.

2.6.4 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations. We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 2.12 Result of the Cartesian product *instructor* × *teaches*.

A Cartesian product of database relations differs in its definition slightly from the mathematical definition of a Cartesian product of sets. Instead of $r_1 \times r_2$ producing pairs (t_1, t_2) of tuples from r_1 and r_2, the relational algebra concatenates t_1 and t_2 into a single tuple, as shown in Figure 2.12.

Since the same attribute name may appear in the schemas of both r_1 and r_2, we need to devise a naming schema to distinguish between these attributes. We do so here by attaching to an attribute the name of the relation from which the attribute originally came. For example, the relation schema for $r = instructor \times teaches$ is:

(*instructor.ID*, *instructor.name*, *instructor.dept_name*, *instructor.salary*,
teaches.ID, *teaches.course_id*, *teaches.sec_id*, *teaches.semester*, *teaches.year*)

Telegram: @uni_k

www.konkur.in

With this schema, we can distinguish *instructor.ID* from *teaches.ID*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as:

(*instructor.ID*, *name*, *dept_name*, *salary*,
teaches.ID, *course_id*, *sec_id*, *semester*, *year*)

This naming convention *requires* that the relations that are the arguments of the Cartesian-product operation have distinct names. This requirement causes problems in some cases, such as when the Cartesian product of a relation with itself is desired. A similar problem arises if we use the result of a relational-algebra expression in a Cartesian product, since we shall need a name for the relation so that we can refer to the relation's attributes. In Section 2.6.8, we see how to avoid these problems by using the rename operation.

Now that we know the relation schema for $r = instructor \times teaches$, what tuples appear in r? As you may suspect, we construct a tuple of r out of each possible pair of tuples: one from the *instructor* relation (Figure 2.1) and one from the *teaches* relation (Figure 2.7). Thus, r is a large relation, as you can see from Figure 2.12, which includes only a portion of the tuples that make up r.

Assume that we have n_1 tuples in *instructor* and n_2 tuples in *teaches*. Then, there are $n_1 * n_2$ ways of choosing a pair of tuples—one tuple from each relation; so there are $n_1 * n_2$ tuples in r. In particular for our example, for some tuples t in r, it may be that the two ID values, *instructor.ID* and *teaches.ID*, are different.

In general, if we have relations $r_1(R_1)$ and $r_2(R_2)$, then $r_1 \times r_2$ is a relation $r(R)$ whose schema R is the concatenation of the schemas R_1 and R_2. Relation r contains all tuples t for which there is a tuple t_1 in r_1 and a tuple t_2 in r_2 for which t and t_1 have the same value on the attributes in R_1 and t and t_2 have the same value on the attributes in R_2.

2.6.5 The Join Operation

Suppose we want to find the information about all instructors together with the *course_id* of all courses they have taught. We need the information in both the *instructor* relation and the *teaches* relation to compute the required result. The Cartesian product of *instructor* and *teaches* does bring together information from both these relations, but unfortunately the Cartesian product associates every instructor with every course that was taught, regardless of whether that instructor taught that course.

Since the Cartesian-product operation associates *every* tuple of *instructor* with every tuple of *teaches*, we know that if an instructor has taught a course (as recorded in the *teaches* relation), then there is some tuple in $instructor \times teaches$ that contains her name and satisfies $instructor.ID = teaches.ID$. So, if we write:

$$\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$$

we get only those tuples of $instructor \times teaches$ that pertain to instructors and the courses that they taught.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Figure 2.13 Result of $\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$.

The result of this expression is shown in Figure 2.13. Observe that instructors Gold, Califieri, and Singh do not teach any course (as recorded in the *teaches* relation), and therefore do not appear in the result.

Note that this expression results in the duplication of the instructor's ID. This can be easily handled by adding a projection to eliminate the column *teaches.ID*.

The *join* operation allows us to combine a selection and a Cartesian product into a single operation.

Consider relations $r(R)$ and $s(S)$, and let θ be a predicate on attributes in the schema $R \cup S$. The **join** operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

Thus, $\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$ can equivalently be written as $instructor \bowtie_{instructor.ID = teaches.ID} teaches$.

2.6.6 Set Operations

Consider a query to find the set of all courses taught in the Fall 2017 semester, the Spring 2018 semester, or both. The information is contained in the *section* relation (Figure 2.6). To find the set of all courses taught in the Fall 2017 semester, we write:

$$\Pi_{course_id}\ (\sigma_{semester = \text{“Fall”} \wedge year = 2017}\ (section))$$

To find the set of all courses taught in the Spring 2018 semester, we write:

$$\Pi_{course_id}\ (\sigma_{semester = \text{“Spring”} \wedge}\qquad\qquad (section))$$

Telegram: @uni_k

www.konkur.in

To answer the query, we need the **union** of these two sets; that is, we need all *course_id*s that appear in either or both of the two relations. We find these data by the binary operation union, denoted, as in set theory, by ∪. So the expression needed is:

$$\Pi_{course_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2017}\ (section)) \cup \Pi_{course_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2018}\ (section))$$

The result relation for this query appears in Figure 2.14. Notice that there are eight tuples in the result, even though there are three distinct courses offered in the Fall 2017 semester and six distinct courses offered in the Spring 2018 semester. Since relations are sets, duplicate values such as CS-101, which is offered in both semesters, are replaced by a single occurrence.

Observe that, in our example, we took the union of two sets, both of which consisted of *course_id* values. In general, for a union operation to make sense:

1. We must ensure that the input relations to the union operation have the same number of attributes; the number of attributes of a relation is referred to as its **arity**.
2. When the attributes have associated types, the types of the *i*th attributes of both input relations must be the same, for each *i*.

Such relations are referred to as **compatible** relations.

For example, it would not make sense to take the union of the *instructor* and *section* relations, since they have different numbers of attributes. And even though the *instructor* and the *student* relations both have arity 4, their 4th attributes, namely, *salary* and *tot_cred*, are of two different types. The union of these two attributes would not make sense in most situations.

The **intersection** operation, denoted by ∩, allows us to find tuples that are in both the input relations. The expression $r \cap s$ produces a relation containing those tuples in

CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 2.14 Courses offered in either Fall 2017, Spring 2018, or both semesters.

Telegram: @uni_k

www.konkur.in

[redacted]
CS-101

Figure 2.15 Courses offered in both the Fall 2017 and Spring 2018 semesters.

r as well as in s. As with the union operation, we must ensure that intersection is done between compatible relations.

Suppose that we wish to find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters. Using set intersection, we can write

$$\Pi_{course_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2017}\ (section)) \cap \Pi_{course_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2018}\ (section))$$

The result relation for this query appears in Figure 2.15.

The **set-difference** operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another. The expression $r - s$ produces a relation containing those tuples in r but not in s.

We can find all the courses taught in the Fall 2017 semester but not in Spring 2018 semester by writing:

$$\Pi_{course_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2017}\ (section)) - \Pi_{course_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2018}\ (section))$$

The result relation for this query appears in Figure 2.16.

As with the union operation, we must ensure that set differences are taken between compatible relations.

2.6.7 The Assignment Operation

It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables. The **assignment** operation, denoted by $\leftarrow$, works like assignment in a programming language. To illustrate this operation, consider the query to find courses that run in Fall 2017 as well as Spring 2018, which we saw earlier. We could write it as:

[redacted]
CS-347
PHY-101

Figure 2.16 Courses offered in the Fall 2017 semester but not in Spring 2018 semester.

Telegram: @uni_k

www.konkur.in

$$courses_fall_2017 \leftarrow \Pi_{course_id}(\sigma_{semester = \text{“Fall”} \wedge year=2017}(section))$$
$$courses_spring_2018 \leftarrow \Pi_{course_id}(\sigma_{semester = \text{“Spring”} \wedge year=2018}(section))$$
$$courses_fall_2017 \cap courses_spring_2018$$

The final line above displays the query result. The preceding two lines assign the query result to a temporary relation. The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the $\leftarrow$ is assigned to the relation variable on the left of the $\leftarrow$. This relation variable may be used in subsequent expressions.

With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query. For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations constitute a database modification. Note that the assignment operation does not provide any additional power to the algebra. It is, however, a convenient way to express complex queries.

2.6.8 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful in some cases to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression E, the expression

$$\rho_x (E)$$

returns the result of expression E under the name x.

A relation r by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name. Some queries require the same relation to be used more than once in the query; in such cases, the rename operation can be used to give unique names to the different occurrences of the same relation.

A second form of the rename operation is as follows: Assume that a relational-algebra expression E has arity n. Then, the expression

$$\rho_{x(A_1,A_2,\ldots,A_n)} (E)$$

returns the result of expression E under the name x, and with the attributes renamed to $A_1, A_2, \ldots, A_n$. This form of the rename operation can be used to give names to attributes in the results of relational algebra operations that involve expressions on attributes.

To illustrate renaming a relation, we consider the query “Find the ID and name of those instructors who earn more than the instructor whose ID is 12121.” (That’s the instructor Wu in the example table in Figure 2.1.)

There are several strategies for writing this query, but to illustrate the rename operation, our strategy is to compare the salary of each instructor with the salary of the

Telegram: @uni_k

www.konkur.in



instructor with ID 12121. The difficulty here is that we need to reference the *instructor* relation once to get the salary of each instructor and then a second time to get the salary of instructor 12121; and we want to do all this in one expression. The rename operator allows us to do this using different names for each referencing of the *instructor* relation. In this example, we shall use the name *i* to refer to our scan of the *instructor* relation in which we are seeking those that will be part of the answer, and *w* to refer to the scan of the *instructor* relation to obtain the salary of instructor 12121:

$$\Pi_{i.ID,i.name}\ ((\sigma_{i.salary\ >\ w.salary}(\rho_i(instructor)\ \times\ \sigma_{w.id=12121}(\rho_w\ (instructor)))))$$

The rename operation is not strictly required, since it is possible to use a positional notation for attributes. We can name attributes of a relation implicitly by using a positional notation, where \$1, \$2, … refer to the first attribute, the second attribute, and so on. The positional notation can also be used to refer to attributes of the results of relational-algebra operations. However, the positional notation is inconvenient for humans, since the position of the attribute is a number, rather than an easy-to-remember attribute name. Hence, we do not use the positional notation in this textbook.

Telegram: @uni_k

www.konkur.in

2.6.9 Equivalent Queries

Note that there is often more than one way to write a query in relational algebra. Consider the following query, which finds information about courses taught by instructors in the Physics department:

$$\sigma_{dept_name = \text{“Physics”}}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

Now consider an alternative query:

$$(\sigma_{dept_name = \text{“Physics”}}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$

Note the subtle difference between the two queries: in the first query, the selection that restricts *dept_name* to Physics is applied after the join of *instructor* and *teaches* has been computed, whereas in the second query, the selection that restricts *dept_name* to Physics is applied to *instructor*, and the join operation is applied subsequently.

Although the two queries are not identical, they are in fact **equivalent**; that is, they give the same result on any database.

Query optimizers in database systems typically look at what result an expression computes and find an efficient way of computing that result, rather than following the exact sequence of steps specified in the query. The algebraic structure of relational algebra makes it easy to find efficient but equivalent alternative expressions, as we will see in Chapter 16.

2.7 Summary

- The relational data model is based on a collection of tables. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations.
- The schema of a relation refers to its logical design, while an instance of the relation refers to its contents at a point in time. The schema of a database and an instance of a database are similarly defined. The schema of a relation includes its attributes, and optionally the types of the attributes and constraints on the relation such as primary and foreign-key constraints.
- A superkey of a relation is a set of one or more attributes whose values are guaranteed to identify tuples in the relation uniquely. A candidate key is a minimal superkey, that is, a set of attributes that forms a superkey, but none of whose subsets is a superkey. One of the candidate keys of a relation is chosen as its primary key.
- A foreign-key constraint from attribute(s) A of relation r_1 to the primary-key B of relation r_2 states that the value of A for each tuple in r_1 must also be the value of B for some tuple in r_2. The relation r_1 is called the referencing relation, and r_2 is called the referenced relation.

Telegram: @uni_k

www.konkur.in

- A schema diagram is a pictorial depiction of the schema of a database that shows the relations in the database, their attributes, and primary keys and foreign keys.
- The relational query languages define a set of operations that operate on tables and output tables as their results. These operations can be combined to get expressions that express desired queries.
- The relational algebra provides a set of operations that take one or more relations as input and return a relation as an output. Practical query languages such as SQL are based on the relational algebra, but they add a number of useful syntactic features.
- The relational algebra defines a set of algebraic operations that operate on tables, and output tables as their results. These operations can be combined to get expressions that express desired queries. The algebra defines the basic operations used within relational query languages like SQL.

Review Terms

- Table
- Relation
- Tuple
- Attribute
- Relation instance
- Domain
- Atomic domain
- Null value
- Database schema
- Database instance
- Relation schema
- Keys
  - Superkey
  - Candidate key
  - Primary key
  - Primary key constraints
- Foreign-key constraint
  - Referencing relation
  - Referenced relation
- Referential integrity constraint
- Schema diagram
- Query language types
  - Imperative
  - Functional
  - Declarative
- Relational algebra
- Relational-algebra expression
- Relational-algebra operations
  - Select σ
  - Project Π
  - Cartesian product $\times$
  - Join $\bowtie$
  - Union $\cup$
  - Set difference $-$
  - Set intersection $\cap$
  - Assignment $\leftarrow$
  - Rename ρ

Telegram: @uni_k

www.konkur.in

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)

Figure 2.17 Employee database.

Practice Exercises

2.1 Consider the employee database of Figure 2.17. What are the appropriate primary keys?

2.2 Consider the foreign-key constraint from the *dept_name* attribute of *instructor* to the *department* relation. Give examples of inserts and deletes to these relations that can cause a violation of the foreign-key constraint.

2.3 Consider the *time_slot* relation. Given that a particular time slot can meet more than once in a week, explain why *day* and *start_time* are part of the primary key of this relation, while *end_time* is not.

2.4 In the instance of *instructor* shown in Figure 2.1, no two instructors have the same name. From this, can we conclude that *name* can be used as a superkey (or primary key) of *instructor*?

2.5 What is the result of first performing the Cartesian product of *student* and *advisor*, and then performing a selection operation on the result with the predicate *s_id* = ID? (Using the symbolic notation of relational algebra, this query can be written as $\sigma_{s_id=ID}(student \times advisor)$.)

2.6 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the name of each employee who lives in city "Miami".

b. Find the name of each employee whose salary is greater than $100000.

c. Find the name of each employee who lives in "Miami" and whose salary is greater than $100000.

2.7 Consider the bank database of Figure 2.18. Give an expression in the relational algebra for each of the following queries:

a. Find the name of each branch located in "Chicago".

b. Find the ID of each borrower who has a loan in branch "Downtown".

Telegram: @uni_k

www.konkur.in

branch(*branch_name*, *branch_city, assets*)
customer (*ID*, *customer_name*, *customer_street, customer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*ID*, *loan_number*)
account (*account_number*, *branch_name, balance*)
depositor (*ID*, *account_number*)

Figure 2.18 Bank database.

2.8 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the ID and name of each employee who does not work for "BigBank".

b. Find the ID and name of each employee who earns at least as much as every employee in the database.

2.9 The **division operator** of relational algebra, "÷", is defined as follows. Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$; that is, every attribute of schema S is also in schema R. Given a tuple t, let $t[S]$ denote the projection of tuple t on the attributes in S. Then $r \div s$ is a relation on schema $R - S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple t is in $r \div s$ if and only if both of two conditions hold:

- t is in $\Pi_{R-S}(r)$
- For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:

  a. $t_r[S] = t_s[S]$

  b. $t_r[R - S] = t$

Given the above definition:

a. Write a relational algebra expression using the division operator to find the IDs of all students who have taken all Comp. Sci. courses. (Hint: project *takes* to just ID and *course_id*, and generate the set of all Comp. Sci. *course_id*s using a select expression, before doing the division.)

b. Show how to write the above query in relational algebra, without using division. (By doing so, you would have shown how to define the division operation using the other relational algebra operations.)

Telegram: @uni_k

www.konkur.in

Exercises

2.10 Describe the differences in meaning between the terms *relation* and *relation schema*.

2.11 Consider the *advisor* relation shown in the schema diagram in Figure 2.9, with *s_id* as the primary key of *advisor*. Suppose a student can have more than one advisor. Then, would *s_id* still be a primary key of the *advisor* relation? If not, what should the primary key of *advisor* be?

2.12 Consider the bank database of Figure 2.18. Assume that branch names and customer names uniquely identify branches and customers, but loans and accounts can be associated with more than one customer.

a. What are the appropriate primary keys?

b. Given your choice of primary keys, identify appropriate foreign keys.

2.13 Construct a schema diagram for the bank database of Figure 2.18.

2.14 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the ID and name of each employee who works for "BigBank".

b. Find the ID, name, and city of residence of each employee who works for "BigBank".

c. Find the ID, name, street address, and city of residence of each employee who works for "BigBank" and earns more than $10000.

d. Find the ID and name of each employee in this database who lives in the same city as the company for which she or he works.

2.15 Consider the bank database of Figure 2.18. Give an expression in the relational algebra for each of the following queries:

a. Find each loan number with a loan amount greater than $10000.

b. Find the ID of each depositor who has an account with a balance greater than $6000.

c. Find the ID of each depositor who has an account with a balance greater than $6000 at the "Uptown" branch.

2.16 List two reasons why null values might be introduced into a database.

2.17 Discuss the relative merits of imperative, functional, and declarative languages.

2.18 Write the following queries in relational algebra, using the university schema.

a. Find the ID and name of each instructor in the Physics department.

Telegram: @uni_k

www.konkur.in

b. Find the ID and name of each instructor in a department located in the building "Watson".

c. Find the ID and name of each student who has taken at least one course in the "Comp. Sci." department.

d. Find the ID and name of each student who has taken at least one course section in the year 2018.

e. Find the ID and name of each student who has not taken any course section in the year 2018.

Further Reading

E. F. Codd of the IBM San Jose Research Laboratory proposed the relational model in the late 1960s ([Codd (1970)]). In that paper, Codd also introduced the original definition of relational algebra. This work led to the prestigious ACM Turing Award to Codd in 1981 ([Codd (1982)]).

After E. F. Codd introduced the relational model, an expansive theory developed around the relational model pertaining to schema design and the expressive power of various relational languages. Several classic texts cover relational database theory, including [Maier (1983)] (which is available free, online), and [Abiteboul et al. (1995)].

Codd's original paper inspired several research projects that were formed in the mid to late 1970s with the goal of constructing practical relational database systems, including System R at the IBM San Jose Research Laboratory, Ingres at the University of California at Berkeley, and Query-by-Example at the IBM T. J. Watson Research Center. The Oracle database was developed commercially at the same time.

Many relational database products are now commercially available. These include IBM's DB2 and Informix, Oracle, Microsoft SQL Server, and Sybase and HANA from SAP. Popular open-source relational database systems include MySQL and PostgreSQL. Hive and Spark are widely used systems that support parallel execution of queries across large numbers of computers.

Bibliography

[Abiteboul et al. (1995)] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison Wesley (1995).

[Codd (1970)] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Codd (1982)] E. F. Codd, "The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity", *Communications of the ACM*, Volume 25, Number 2 (1982), pages 109–117.

Telegram: @uni_k

www.konkur.in

[Maier (1983)] D. Maier, *The Theory of Relational Databases*, Computer Science Press (1983).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 3



Introduction to SQL

In this chapter, as well as in Chapter 4 and Chapter 5, we study the most widely used database query language, SQL.

Although we refer to the SQL language as a "query language," it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints.

It is not our intention to provide a complete users' guide for SQL. Rather, we present SQL's fundamental constructs and concepts. Individual implementations of SQL may differ in details or may support only a subset of the full language.

We strongly encourage you to try out the SQL queries that we describe here on an actual database. See the Tools section at the end of this chapter for tips on what database systems you could use, and how to create the schema, populate sample data, and execute your queries.

3.1 Overview of the SQL Query Language

IBM developed the original version of SQL, originally called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language). Many products now support the SQL language. SQL has clearly established itself as *the* standard relational database language.

In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86. ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, followed by SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, and most recently SQL:2016.

The SQL language has several parts:

- **Data-definition language** (**DDL**). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

Telegram: @uni_k

www.konkur.in

- **Data-manipulation language** (**DML**). The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity**. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition**. The SQL DDL includes commands for defining views.
- **Transaction control**. SQL includes commands for specifying the beginning and end points of transactions.
- **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

In this chapter, we present a survey of basic DML and the DDL features of SQL. Features described here have been part of the SQL standard since SQL-92.

In Chapter 4, we provide a more detailed coverage of the SQL query language, including (a) various join expressions, (b) views, (c) transactions, (d) integrity constraints, (e) type system, and (f) authorization.

In Chapter 5, we cover more advanced features of the SQL language, including (a) mechanisms to allow accessing SQL from a programming language, (b) SQL functions and procedures, (c) triggers, (d) recursive queries, (e) advanced aggregation features, and (f) several features designed for data analysis.

Although most SQL implementations support the standard features we describe here, there are differences between implementations. Most implementations support some nonstandard features while omitting support for some of the more advanced and more recent features. In case you find that some language features described here do not work on the database system that you use, consult the user manuals for your database system to find exactly what features it supports.

3.2 SQL Data Definition

The set of relations in a database are specified using a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.

Telegram: @uni_k

www.konkur.in

- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

We discuss here basic schema definition and basic types; we defer discussion of the other SQL DDL features to Chapter 4 and Chapter 5.

3.2.1 Basic Types

The SQL standard supports a variety of built-in types, including:

- **char**(n): A fixed-length character string with user-specified length n. The full form, **character**, can be used instead.
- **varchar**(n): A variable-length character string with user-specified maximum length n. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric**(p, d): A fixed-point number with user-specified precision. The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44.5 to be stored exactly, but neither 444.5 nor 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float**(n): A floating-point number with precision of at least n digits.

Additional types are covered in Section 4.5.

Each type may include a special value called the **null** value. A null value indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered, as we shall see shortly.

The **char** data type stores fixed-length strings. Consider, for example, an attribute A of type **char**(10). If we stored a string "Avi" in this attribute, seven spaces are appended to the string to make it 10 characters long. In contrast, if attribute B were of type **varchar**(10), and we stored "Avi" in attribute B, no spaces would be added. When comparing two values of type **char**, if they are of different lengths, extra spaces are automatically attached to the shorter one to make them the same size before comparison.

When comparing a **char** type with a **varchar** type, one may expect extra spaces to be added to the **varchar** type to make the lengths equal, before comparison; however, this may or may not be done, depending on the database system. As a result, even if

Telegram: @uni_k

www.konkur.in

the same value "Avi" is stored in the attributes A and B above, a comparison $A=B$ may return false. We recommend you always use the **varchar** type instead of the **char** type to avoid these problems.

SQL also provides the **nvarchar** type to store multilingual data using the Unicode representation. However, many databases allow Unicode (in the UTF-8 representation) to be stored even in **varchar** types.

3.2.2 Basic Schema Definition

We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database:

```
create table department
        (dept_name   varchar (20),
         building    varchar (15),
         budget      numeric (12,2),
         primary key (dept_name));
```

The relation created above has three attributes, *dept_name*, which is a character string of maximum length 20, *building*, which is a character string of maximum length 15, and *budget*, which is a number with 12 digits in total, two of which are after the decimal point. The **create table** command also specifies that the *dept_name* attribute is the primary key of the *department* relation.

The general form of the **create table** command is:

```
create table r
        (A1  D1,
         A2  D2,
         ...,
         An  Dn,
         ⟨integrity-constraint1⟩,
         ...,
         ⟨integrity-constraintk⟩);
```

where r is the name of the relation, each A_i is the name of an attribute in the schema of relation r, and D_i is the domain of attribute A_i; that is, D_i specifies the type of attribute A_i along with optional constraints that restrict the set of allowed values for A_i.

The semicolon shown at the end of the **create table** statements, as well as at the end of other SQL statements later in this chapter, is optional in many SQL implementations.

SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

- **primary key** $(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$: The primary-key specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ form the primary key for the relation. The primary-key attributes

Telegram: @uni_k

www.konkur.in

are required to be *nonnull* and *unique*; that is, no tuple can have a null value for a primary-key attribute, and no two tuples in the relation can be equal on all the primary-key attributes. Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation.

- **foreign key** $(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ **references** *s*: The **foreign key** specification says that the values of attributes $(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation *s*.

  Figure 3.1 presents a partial SQL DDL definition of the university database we use in the text. The definition of the *course* table has a declaration "**foreign key** (*dept_name*) **references** *department*". This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the primary key attribute (*dept_name*) of the *department* relation. Without this constraint, it is possible for a course to specify a nonexistent department name. Figure 3.1 also shows foreign-key constraints on tables *section*, *instructor* and *teaches*. Some database systems, including MySQL, require an alternative syntax, "**foreign key** (*dept_name*) **references** *department*(*dept_name*)", where the referenced attributes in the referenced table are listed explicitly.

- **not null**: The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute. For example, in Figure 3.1, the **not null** constraint on the *name* attribute of the *instructor* relation ensures that the name of an instructor cannot be null.

More details on the foreign-key constraint, as well as on other integrity constraints that the **create table** command may include, are provided later, in Section 4.4.

SQL prevents any update to the database that violates an integrity constraint. For example, if a newly inserted or modified tuple in a relation has null values for any primary-key attribute, or if the tuple has the same value on the primary-key attributes as does another tuple in the relation, SQL flags an error and prevents the update. Similarly, an insertion of a *course* tuple with a *dept_name* value that does not appear in the *department* relation would violate the foreign-key constraint on *course*, and SQL prevents such an insertion from taking place.

A newly created relation is empty initially. Inserting tuples into a relation, updating them, and deleting them are done by data manipulation statements **insert**, **update**, and **delete**, which are covered in Section 3.9.

To remove a relation from an SQL database, we use the **drop table** command. The **drop table** command deletes all information about the dropped relation from the database. The command

drop table *r*;

is a more drastic action than

Telegram: @uni_k

www.konkur.in


```
create table department
    (dept_name      varchar (20),
     building       varchar (15),
     budget         numeric (12,2),
     primary key (dept_name));

create table course
    (course_id      varchar (7),
     title          varchar (50),
     dept_name      varchar (20),
     credits        numeric (2,0),
     primary key (course_id),
     foreign key (dept_name) references department);

create table instructor
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     salary         numeric (8,2),
     primary key (ID),
     foreign key (dept_name) references department);

create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course);

create table teaches
    (ID             varchar (5),
     course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section,
     foreign key (ID) references instructor);
```

Figure 3.1 SQL data definition for part of the

Telegram: @uni_k

www.konkur.in

delete from *r*;

The latter retains relation *r*, but deletes all tuples in *r*. The former deletes not only all tuples of *r*, but also the schema for *r*. After *r* is dropped, no tuples can be inserted into *r* unless it is re-created with the **create table** command.

We use the **alter table** command to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **alter table** command is

alter table *r* **add** *A D*;

where *r* is the name of an existing relation, *A* is the name of the attribute to be added, and *D* is the type of the added attribute. We can drop attributes from a relation by the command

alter table *r* **drop** *A*;

where *r* is the name of an existing relation, and *A* is the name of an attribute of the relation. Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

3.3 Basic Structure of SQL Queries

The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**. A query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result. We introduce the SQL syntax through examples, and we describe the general structure of SQL queries later.

3.3.1 Queries on a Single Relation

Let us consider a simple query using our university example, "Find the names of all instructors." Instructor names are found in the *instructor* relation, so we put that relation in the **from** clause. The instructor's name appears in the *name* attribute, so we put that in the **select** clause.

select *name*
from *instructor*;

The result is a relation consisting of a single attribute with the heading *name*. If the *instructor* relation is as shown in Figure 2.1, then the relation that results from the preceding query is shown in Figure 3.2.

Telegram: @uni_k

www.konkur.in

Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of "**select** *name* **from** *instructor*".

Now consider another query, "Find the department names of all instructors," which can be written as:

select *dept_name*
from *instructor*;

Since more than one instructor can belong to a department, a department name could appear more than once in the *instructor* relation. The result of the above query is a relation containing the department names, shown in Figure 3.3.

In the formal, mathematical definition of the relational model, a relation is a set. Thus, duplicate tuples would never appear in relations. In practice, duplicate elimination is time-consuming. Therefore, SQL allows duplicates in database relations as well as in the results of SQL expressions.[1] Thus, the preceding SQL query lists each department name once for every tuple in which it appears in the *instructor* relation.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as:

select distinct *dept_name*
from *instructor*;

if we want duplicates removed. The result of the above query would contain each department name at most once.

[1] Any database relation whose schema includes a primary-key declaration cannot contain duplicate tuples, since they would violate the primary-key constraint.

Telegram: @uni_k

www.konkur.in

Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of "**select** *dept_name* **from** *instructor*".

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

select all *dept_name*
from *instructor*;

Since duplicate retention is the default, we shall not use **all** in our examples. To ensure the elimination of duplicates in the results of our example queries, we shall use **distinct** whenever it is necessary.

The **select** clause may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query:

select *ID*, *name*, *dept_name*, *salary* * 1.1
from *instructor*;

returns a relation that is the same as the *instructor* relation, except that the attribute *salary* is multiplied by 1.1. This shows what would result if we gave a 10% raise to each instructor; note, however, that it does not result in any change to the *instructor* relation.

SQL also provides special data types, such as various forms of the *date* type, and allows several arithmetic functions to operate on these types. We discuss this further in Section 4.5.1.

The **where** clause allows us to select only those rows in the result relation of the **from** clause that satisfy a specified predicate. Consider the query "Find the names of all instructors in the Computer Science department who have salary greater than $70,000." This query can be written in SQL as:

Telegram: @uni_k

www.konkur.in



Figure 3.4 Result of "Find the names of all instructors in the Computer Science department who have salary greater than $70,000."

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000;
```

If the *instructor* relation is as shown in Figure 2.1, then the relation that results from the preceding query is shown in Figure 3.4.

SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>. SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

We shall explore other features of **where** clause predicates later in this chapter.

3.3.2 Queries on Multiple Relations

So far our example queries were on a single relation. Queries often need to access information from multiple relations. We now study how to write such queries.

As an example, suppose we want to answer the query "Retrieve the names of all instructors, along with their department names and department building name."

Looking at the schema of the relation *instructor*, we realize that we can get the department name from the attribute *dept_name*, but the department building name is present in the attribute *building* of the relation *department*. To answer the query, each tuple in the *instructor* relation must be matched with the tuple in the *department* relation whose *dept_name* value matches the *dept_name* value of the *instructor* tuple.

In SQL, to answer the above query, we list the relations that need to be accessed in the **from** clause and specify the matching condition in the **where** clause. The above query can be written in SQL as

```
select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name= department.dept_name;
```

If the *instructor* and *department* relations are as shown in Figure 2.1 and Figure 2.5 respectively, then the result of this query is shown in Figure 3.5.

Note that the attribute *dept_name* occurs in both the relations *instructor* and *department*, and the relation name is used as a prefix (in *instructor.dept name*, and *de-*

Telegram: @uni_k

www.konkur.in

Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 3.5 The result of "Retrieve the names of all instructors, along with their department names and department building name."

partment.dept_name) to make clear to which attribute we are referring. In contrast, the attributes *name* and *building* appear in only one of the relations and therefore do not need to be prefixed by the relation name.

This naming convention *requires* that the relations that are present in the **from** clause have distinct names. This requirement causes problems in some cases, such as when information from two different tuples in the same relation needs to be combined. In Section 3.4.1, we see how to avoid these problems by using the rename operation.

We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause. The role of each clause is as follows:

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the **from** clause.

A typical SQL query has the form:

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P;

Telegram: @uni_k

www.konkur.in

Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the **where** clause is omitted, the predicate P is **true**.

Although the clauses must be written in the order **select**, **from**, **where**, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first **from**, then **where**, and then **select**.[2]

The **from** clause by itself defines a Cartesian product of the relations listed in the clause. It is defined formally in terms of relational algebra, but it can also be understood as an iterative process that generates tuples for the result relation of the **from** clause.

```
for each tuple t1 in relation r1
        for each tuple t2 in relation r2
                ...
                for each tuple tm in relation rm
                        Concatenate t1, t2, ..., tm into a single tuple t
                        Add t into the result relation
```

The result relation has all attributes from all the relations in the **from** clause. Since the same attribute name may appear in both r_i and r_j, as we saw earlier, we prefix the name of the relation from which the attribute originally came, before the attribute name.

For example, the relation schema for the Cartesian product of relations *instructor* and *teaches* is:

(*instructor.ID*, *instructor.name*, *instructor.dept_name*, *instructor.salary*, *teaches.ID*, *teaches.course_id*, *teaches.sec_id*, *teaches.semester*, *teaches.year*)

With this schema, we can distinguish *instructor.ID* from *teaches.ID*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema as:

(*instructor.ID*, *name*, *dept_name*, *salary*, *teaches.ID*, *course_id*, *sec_id*, *semester*, *year*)

To illustrate, consider the *instructor* relation in Figure 2.1 and the *teaches* relation in Figure 2.7. Their Cartesian product is shown in Figure 3.6, which includes only a portion of the tuples that make up the Cartesian product result.

The Cartesian product by itself combines tuples from *instructor* and *teaches* that are unrelated to each other. Each tuple in *instructor* is combined with *every* tuple in *teaches*, even those that refer to a different instructor. The result can be an extremely large relation, and it rarely makes sense to create such a Cartesian product.

[2] In practice, SQL may convert the expression into an equivalent form that can be processed more efficiently. However, we shall defer concerns about efficiency to Chapter 15 and Chapter 16.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 3.6 The Cartesian product of the *instructor* relation with the *teaches* relation.

Instead, the predicate in the **where** clause is used to restrict the combinations created by the Cartesian product to those that are meaningful for the desired answer. We would likely want a query involving *instructor* and *teaches* to combine a particular tuple *t* in *instructor* with only those tuples in *teaches* that refer to the same instructor to which *t* refers. That is, we wish only to match *teaches* tuples with *instructor* tuples that have the same *ID* value. The following SQL query ensures this condition and outputs the instructor name and course identifiers from such matching tuples.

select *name*, *course_id*
from *instructor*, *teaches*
where *instructor.ID*= *teaches.ID*;

Telegram: @uni_k

www.konkur.in

Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.7 Result of "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

Note that the preceding query outputs only instructors who have taught some course. Instructors who have not taught any course are not output; if we wish to output such tuples, we could use an operation called the *outer join*, which is described in Section 4.1.3.

If the *instructor* relation is as shown in Figure 2.1 and the *teaches* relation is as shown in Figure 2.7, then the relation that results from the preceding query is shown in Figure 3.7. Observe that instructors Gold, Califieri, and Singh, who have not taught any course, do not appear in Figure 3.7.

If we wished to find only instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the **where** clause, as shown below.

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID and instructor.dept_name = 'Comp. Sci.';
```

Note that since the *dept_name* attribute occurs only in the *instructor* relation, we could have used just *dept_name*, instead of *instructor.dept_name* in the above query.

In general, the meaning of an SQL query can be understood as follows:

1. Generate a Cartesian product of the relations listed in the **from** clause.
2. Apply the predicates specified in the **where** clause on the result of Step 1.

Telegram: @uni_k

www.konkur.in

3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the **select** clause.

This sequence of steps helps make clear what the result of an SQL query should be, *not* how it should be executed. A real implementation of SQL would not execute the query in this fashion; it would instead optimize evaluation by generating (as far as possible) only elements of the Cartesian product that satisfy the **where** clause predicates. We study such implementation techniques in Chapter 15 and Chapter 16.

When writing queries, you should be careful to include appropriate **where** clause conditions. If you omit the **where** clause condition in the preceding SQL query, it will output the Cartesian product, which could be a huge relation. For the example *instructor* relation in Figure 2.1 and the example *teaches* relation in Figure 2.7, their Cartesian product has $12 * 13 = 156$ tuples—more than we can show in the text! To make matters worse, suppose we have a more realistic number of instructors than we show in our sample relations in the figures, say 200 instructors. Let's assume each instructor teaches three courses, so we have 600 tuples in the *teaches* relation. Then the preceding iterative process generates $200 * 600 = 120{,}000$ tuples in the result.

3.4 Additional Basic Operations

A number of additional basic operations are supported in SQL.

3.4.1 The Rename Operation

Consider again the query that we used earlier:

select *name*, *course_id*
from *instructor*, *teaches*
where *instructor.ID*= *teaches.ID*;

The result of this query is a relation with the following attributes:

name, *course_id*

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

We cannot, however, always derive names in this way, for several reasons: First, two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result. Second, if we use an arithmetic expression in the **select** clause, the resultant attribute does not have a name. Third, even if an attribute name can be derived from the base relations as in the preceding example, we may want to change the attribute name in the result. Hence, SQL provides a way of renaming the attributes of a result relation. It uses the **as clause**, taking the form:

Telegram: @uni_k

www.konkur.in

mul-

tiset relational algebra

1.

2.

3.

Telegram: @uni_k

www.konkur.in

old-name **as** *new-name*

The **as** clause can appear in both the **select** and **from** clauses.[3]

For example, if we want the attribute name *name* to be replaced with the name *instructor_name*, we can rewrite the preceding query as:

select *name* **as** *instructor_name*, *course_id*
from *instructor*, *teaches*
where *instructor.ID*= *teaches.ID*;

The **as** clause is particularly useful in renaming relations. One reason to rename a relation is to replace a long relation name with a shortened version that is more convenient to use elsewhere in the query. To illustrate, we rewrite the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

select *T.name*, *S.course_id*
from *instructor* **as** *T*, *teaches* **as** *S*
where *T.ID*= *S.ID*;

Another reason to rename a relation is a case where we wish to compare tuples in the same relation. We then need to take the Cartesian product of a relation with itself and, without renaming, it becomes impossible to distinguish one tuple from the other. Suppose that we want to write the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department." We can write the SQL expression:

select distinct *T.name*
from *instructor* **as** *T*, *instructor* **as** *S*
where *T.salary* > *S.salary* **and** *S.dept_name* = 'Biology';

Observe that we could not use the notation *instructor.salary*, since it would not be clear which reference to *instructor* is intended.

In the above query, *T* and *S* can be thought of as copies of the relation *instructor*, but more precisely, they are declared as aliases, that is, as alternative names, for the relation *instructor*. An identifier, such as *T* and *S*, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but it is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

[3]Early versions of SQL did not include the keyword **as**. As a result, some implementations of SQL, notably Oracle, do not permit the keyword **as** in the **from** clause. In Oracle, "*old-name* **as** *new-name*" is written instead as "*old-name new-name*" in the **from** clause. The keyword **as** is permitted for renaming attributes in the **select** clause, but it is optional and may be omitted in Oracle.

Telegram: @uni_k

www.konkur.in

Note that a better way to phrase the previous query in English would be "Find the names of all instructors who earn more than the lowest paid instructor in the Biology department." Our original wording fits more closely with the SQL that we wrote, but the latter wording is more intuitive, and it can in fact be expressed directly in SQL as we shall see in Section 3.8.2.

3.4.2 String Operations

SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. A single quote character that is part of a string can be specified by using two single quote characters; for example, the string "It's right" can be specified by 'It''s right'.

The SQL standard specifies that the equality operation on strings is case sensitive; as a result, the expression "'comp. sci.' = 'Comp. Sci.'" evaluates to false. However, some database systems, such as MySQL and SQL Server, do not distinguish uppercase from lowercase when matching strings; as a result, "'comp. sci.' = 'Comp. Sci.'" would evaluate to true on these systems. This default behavior can, however, be changed, either at the database level or at the level of specific attributes.

SQL also permits a variety of functions on character strings, such as concatenating (using "||"), extracting substrings, finding the length of strings, converting strings to uppercase (using the function **upper**(s) where s is a string) and lowercase (using the function **lower**(s)), removing spaces at the end of the string (using **trim**(s)), and so on. There are variations on the exact set of string functions supported by different database systems. See your database system's manual for more details on exactly what string functions it supports.

Pattern matching can be performed on strings using the operator **like**. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore (_): The _ character matches any character.

Patterns are case sensitive; [4] that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Intro%' matches any string beginning with "Intro".
- '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
- '_ _ _' matches any string of exactly three characters.
- '_ _ _%' matches any string of at least three characters.

[4] Except for MySQL, or with the **ilike** operator in PostgreSQL, where patterns

Telegram: @uni_k

www.konkur.in

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all departments whose building name includes the substring 'Watson'." This query can be written as:

```
select dept_name
from department
where building like '%Watson%';
```

For patterns to include the special pattern characters (that is, % and _), SQL allows the specification of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a **like** comparison using the **escape** keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

- **like** 'ab\%cd%' **escape** '\' matches all strings beginning with "ab%cd".
- **like** 'ab\\cd%' **escape** '\' matches all strings beginning with "ab\cd".

SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator. Some implementations provide variants of the **like** operation that do not distinguish lower- and uppercase.

Some SQL implementations, notably PostgreSQL, offer a **similar to** operation that provides more powerful pattern matching than the **like** operation; the syntax for specifying patterns is similar to that used in Unix regular expressions.

3.4.3 Attribute Specification in the Select Clause

The asterisk symbol " * " can be used in the **select** clause to denote "all attributes." Thus, the use of *instructor*.* in the **select** clause of the query:

```
select instructor.*
from instructor, teaches
where instructor.ID= teaches.ID;
```

indicates that all attributes of *instructor* are to be selected. A **select** clause of the form **select** * indicates that all attributes of the result relation of the **from** clause are selected.

3.4.4 Ordering the Display of Tuples

SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

Telegram: @uni_k

www.konkur.in


```
select name
from instructor
where dept_name = 'Physics'
order by name;
```

By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire *instructor* relation in descending order of *salary*. If several instructors have the same salary, we order them in ascending order by name. We express this query in SQL as follows:

```
select *
from instructor
order by salary desc, name asc;
```

3.4.5 Where-Clause Predicates

SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. If we wish to find the names of instructors with salary amounts between $90,000 and $100,000, we can use the **between** comparison to write:

```
select name
from instructor
where salary between 90000 and 100000;
```

instead of:

```
select name
from instructor
where salary <= 100000 and salary >= 90000;
```

Similarly, we can use the **not between** comparison operator.

SQL permits us to use the notation $(v_1, v_2, \ldots, v_n)$ to denote a tuple of arity n containing values $v_1, v_2, \ldots, v_n$; the notation is called a *row constructor*. The comparison operators can be used on tuples, and the ordering is defined lexicographically. For example, (a_1, a_2) <= (b_1, b_2) is true if a_1 <= b_1 **and** a_2 <= b_2; similarly, the two tuples are equal if all their attributes are equal. Thus, the SQL query:

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID and dept_name = 'Biology';
```

Telegram: @uni_k

www.konkur.in

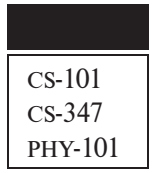


Figure 3.8 The *c1* relation, listing courses taught in Fall 2017.

can be rewritten as follows:[5]

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

3.5 Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set operations ∪, ∩, and −. We shall now construct queries involving the **union**, **intersect**, and **except** operations over two sets.

- The set of all courses taught in the Fall 2017 semester:

```
select course_id
from section
where semester = 'Fall' and year= 2017;
```

- The set of all courses taught in the Spring 2018 semester:

```
select course_id
from section
where semester = 'Spring' and year= 2018;
```

In our discussion that follows, we shall refer to the relations obtained as the result of the preceding queries as *c1* and *c2*, respectively, and show the results when these queries are run on the *section* relation of Figure 2.6 in Figure 3.8 and Figure 3.9. Observe that *c2* contains two tuples corresponding to *course_id* CS-319, since two sections of the course were offered in Spring 2018.

[5] Although it is part of the SQL-92 standard, some SQL

Telegram: @uni_k

www.konkur.in

CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Figure 3.9 The *c2* relation, listing courses taught in Spring 2018.

3.5.1 The Union Operation

To find the set of all courses taught either in Fall 2017 or in Spring 2018, or both, we write the following query. Note that the parentheses we include around each **select-from-where** statement below are optional but useful for ease of reading; some databases do not allow the use of the parentheses, in which case they may be dropped.

```
(select course_id
 from section
 where semester = 'Fall' and year= 2017)
union
(select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

The **union** operation automatically eliminates duplicates, unlike the **select** clause. Thus, using the *section* relation of Figure 2.6, where two sections of CS-319 are offered in Spring 2018, and a section of CS-101 is offered in the Fall 2017 as well as in the Spring 2018 semesters, CS-101 and CS-319 appear only once in the result, shown in Figure 3.10.

If we want to retain all duplicates, we must write **union all** in place of **union**:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2017)
union all
(select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both *c1* and *c2*. So, in the above query, each of -319 and -101 would

Telegram: @uni_k

www.konkur.in

CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 3.10 The result relation for *c1* union *c2*.

be listed twice. As a further example, if it were the case that four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, then there would be six tuples with ECE-101 in the result.

3.5.2 The Intersect Operation

To find the set of all courses taught in both the Fall 2017 and Spring 2018, we write:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2017)
intersect
(select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

The result relation, shown in Figure 3.11, contains only one tuple with CS-101. The **intersect** operation automatically eliminates duplicates. [6] For example, if it were the case that four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, then there would be only one tuple with ECE-101 in the result.

CS-101

Figure 3.11 The result relation for *c1* intersect *c2*.

[6] MySQL does not implement the **intersect** operation; a work-around is to use subqueries as discussed in Section 3.8.1.

Telegram: @uni_k

www.konkur.in

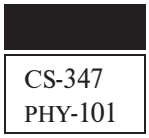


Figure 3.12 The result relation for *c1* except *c2*.

If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2017)
intersect all
(select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both *c1* and *c2*. For example, if four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, then there would be two tuples with ECE-101 in the result.

3.5.3 The Except Operation

To find all courses taught in the Fall 2017 semester but not in the Spring 2018 semester, we write:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2017)
except
(select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

The result of this query is shown in Figure 3.12. Note that this is exactly relation *c1* of Figure 3.8 except that the tuple for CS-101 does not appear. The **except** operation [7] outputs all tuples from its first input that do not occur in the second input; that is, it

[7] Some SQL implementations, notably Oracle, use the keyword **minus** in place of **except**, while Oracle 12c uses the keywords **multiset except** in place of **except all**. MySQL does not implement it at all; a work-around is to use subqueries as discussed in Section 3.8.1.

Telegram: @uni_k

www.konkur.in

performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference. For example, if four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, the result of the **except** operation would not have any copy of ECE-101.

If we want to retain duplicates, we must write **except all** in place of **except**:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2017)
except all
(select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies in *c1* minus the number of duplicate copies in *c2*, provided that the difference is positive. Thus, if four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in Spring 2018, then there are two tuples with ECE-101 in the result. If, however, there were two or fewer sections of ECE-101 in the Fall 2017 semester and two sections of ECE-101 in the Spring 2018 semester, there is no tuple with ECE-101 in the result.

3.6 Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

The result of an arithmetic expression (involving, for example, $+$, $-$, $*$, or $/$) is null if any of the input values is null. For example, if a query has an expression $r.A + 5$, and $r.A$ is null for a particular tuple, then the expression result must also be null for that tuple.

Comparisons involving nulls are more of a problem. For example, consider the comparison "$1 <$ **null**". It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, "**not** $(1 <$ **null**$)$" would evaluate to true, which does not make sense. SQL therefore treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**, which are described later in this section). This creates a third logical value in addition to *true* and *false*.

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.

Telegram: @uni_k

www.konkur.in

- **and**: The result of *true* **and** *unknown* is *unknown*, *false* **and** *unknown* is *false*, while *unknown* **and** *unknown* is *unknown*.
- **or**: The result of *true* **or** *unknown* is *true*, *false* **or** *unknown* is *unknown*, while *unknown* **or** *unknown* is *unknown*.
- **not**: The result of **not** *unknown* is *unknown*.

You can verify that if $r.A$ is null, then "$1 < r.A$" as well as "**not** $(1 < r.A)$" evaluate to unknown.

If the **where** clause predicate evaluates to either **false** or **unknown** for a tuple, that tuple is not added to the result.

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the *instructor* relation with null values for *salary*, we write:

select *name*
from *instructor*
where *salary* **is null**;

The predicate **is not null** succeeds if the value on which it is applied is not null.

SQL allows us to test whether the result of a comparison is unknown, rather than true or false, by using the clauses **is unknown** and **is not unknown**.[8] For example,

select *name*
from *instructor*
where *salary* $>$ 10000 **is unknown**;

When a query uses the **select distinct** clause, duplicate tuples must be eliminated. For this purpose, when comparing values of corresponding attributes from two tuples, the values are treated as identical if either both are non-null and equal in value, or both are null. Thus, two copies of a tuple, such as {('A',null), ('A',null)}, are treated as being identical, even if some of the attributes have a null value. Using the **distinct** clause then retains only one copy of such identical tuples. Note that the treatment of null above is different from the way nulls are treated in predicates, where a comparison "null=null" would return unknown, rather than true.

The approach of treating tuples as identical if they have the same values for all attributes, even if some of the values are null, is also used for the set operations union, intersection, and except.

[8]The **is unknown** and **is not unknown** constructs are not supported by several

Telegram: @uni_k

www.konkur.in

3.7 Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five standard built-in aggregate functions:[9]

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

3.7.1 Basic Aggregation

Consider the query "Find the average salary of instructors in the Computer Science department." We write this query as follows:

```
select avg (salary)
from instructor
where dept_name = 'Comp. Sci.';
```

The result of this query is a relation with a single attribute containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department. The database system may give an awkward name to the result relation attribute that is generated by aggregation, consisting of the text of the expression; however, we can give a meaningful name to the attribute by using the **as** clause as follows:

```
select avg (salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.';
```

In the *instructor* relation of Figure 2.1, the salaries in the Computer Science department are \$75,000, \$65,000, and \$92,000. The average salary is \$232,000/3 = \$77,333.33.

Retaining duplicates is important in computing an average. Suppose the Computer Science department adds a fourth instructor whose salary happens to be \$75,000. If du-

[9] Most implementations of SQL offer a number of additional

Telegram: @uni_k

www.konkur.in

plicates were eliminated, we would obtain the wrong answer ($232,000/4 = $58,000) rather than the correct answer of $76,750.

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query "Find the total number of instructors who teach a course in the Spring 2018 semester." In this case, an instructor counts only once, regardless of the number of course sections that the instructor teaches. The required information is contained in the relation *teaches*, and we write this query as follows:

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

Because of the keyword **distinct** preceding *ID*, even if an instructor teaches more than one course, she is counted only once in the result.

We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count** (*). Thus, to find the number of tuples in the *course* relation, we write

```
select count (*)
from course;
```

SQL does not allow the use of **distinct** with **count** (*). It is legal to use **distinct** with **max** and **min**, even though the result does not change. We can use the keyword **all** in place of **distinct** to specify duplicate retention, but since **all** is the default, there is no need to do so.

3.7.2 Aggregation with Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;
```

Telegram: @uni_k

www.konkur.in

76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Figure 3.13 Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

Figure 3.13 shows the tuples in the *instructor* relation grouped by the *dept_name* attribute, which is the first step in computing the query result. The specified aggregate is computed for each group, and the result of the query is shown in Figure 3.14.

In contrast, consider the query "Find the average salary of all instructors." We write this query as follows:

select avg (*salary*)
from *instructor*;

In this case the **group by** clause has been omitted, so the entire relation is treated as a single group.

Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.14 The result relation for the query "Find the average salary in each department".

Telegram: @uni_k

www.konkur.in

As another example of aggregation on groups of tuples, consider the query "Find the number of instructors in each department who teach a course in the Spring 2018 semester." Information about which instructors teach which course sections in which semester is available in the *teaches* relation. However, this information has to be joined with information from the *instructor* relation to get the department name of each instructor. Thus, we write this query as follows:

```
select dept_name, count (distinct ID) as instr_count
from instructor, teaches
where instructor.ID= teaches.ID and
      semester = 'Spring' and year = 2018
group by dept_name;
```

The result is shown in Figure 3.15.

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause. In other words, any attribute that is not present in the **group by** clause may appear in the **select** clause only as an argument to an aggregate function, otherwise the query is treated as erroneous. For example, the following query is erroneous since *ID* does not appear in the **group by** clause, and yet it appears in the **select** clause without being aggregated:

```
/* erroneous query */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

In the preceding query, each instructor in a particular group (defined by *dept_name*) can have a different *ID*, and since only one tuple is output for each group, there is no unique way of choosing which *ID* value to output. As a result, such cases are disallowed by SQL.

The preceding query also illustrates a comment written in SQL by enclosing text in "/* */"; the same comment could have also been written as "-- erroneous query".

Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.15 The result relation for the query "Find the number of instructors in each department who teach a course in the Spring 2018 semester."

Telegram: @uni_k

www.konkur.in

Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.16 The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

3.7.3 The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used in the **having** clause. We express this query in SQL as follows:

select *dept_name*, **avg** (*salary*) **as** *avg_salary*
from *instructor*
group by *dept_name*
having avg (*salary*) > 42000;

The result is shown in Figure 3.16.

As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is erroneous.

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

Telegram: @uni_k

www.konkur.in

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "For each course section offered in 2017, find the average total credits (*tot_cred*) of all students enrolled in the section, if the section has at least 2 students."

```
select course_id, semester, year, sec_id, avg (tot_cred)
from student, takes
where student.ID= takes.ID and year = 2017
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Note that all the required information for the preceding query is available from the relations *takes* and *student*, and that although the query pertains to sections, a join with *section* is not needed.

3.7.4 Aggregation with Null and Boolean Values

Null values, when they exist, complicate the processing of aggregate operators. For example, assume that some tuples in the *instructor* relation have a null value for *salary*. Consider the following query to total all salary amounts:

```
select sum (salary)
from instructor;
```

The values to be summed in the preceding query include null values, since we assumed that some tuples have a null value for *salary*. Rather than say that the overall sum is itself *null*, the SQL standard says that the **sum** operator should ignore *null* values in its input.

In general, aggregate functions treat nulls according to the following rule: All aggregate functions except **count (*)** ignore null values in their input collection. As a result of null values being ignored, the collection of values may be empty. The **count** of an empty collection is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection. The effect of null values on some of the more complicated SQL constructs can be subtle.

A **Boolean** data type that can take values **true**, **false**, and **unknown** was introduced in SQL:1999. The aggregate functions **some** and **every** can be applied on a collection of Boolean values, and compute the disjunction (**or**) and conjunction (**and**), respectively, of the values.

Telegram: @uni_k

www.konkur.in



Telegram: @uni_k

www.konkur.in

3.8 Nested Subqueries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality by nesting subqueries in the **where** clause. We study such uses of nested subqueries in the **where** clause in Section 3.8.1 through Section 3.8.4. In Section 3.8.5, we study nesting of subqueries in the **from** clause. In Section 3.8.7, we see how a class of subqueries called scalar subqueries can appear wherever an expression returning a value can occur.

3.8.1 Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

As an illustration, reconsider the query "Find all the courses taught in the both the Fall 2017 and Spring 2018 semesters." Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2017 and the set of courses taught in Spring 2018. We can take the alternative approach of finding all courses that were taught in Fall 2017 and that are also members of the set of courses taught in Spring 2018. This formulation generates the same results as the previous one did, but it leads us to write our query using the **in** connective of SQL. We begin by finding all courses taught in Spring 2018, and we write the subquery:

```
(select course_id
 from section
 where semester = 'Spring' and year= 2018)
```

We then need to find those courses that were taught in the Fall 2017 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the **where** clause of an outer query. The resulting query is:

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
      course_id in (select course_id
                    from section
                    where semester = 'Spring' and year= 2018);
```

Note that we need to use **distinct** here because the **intersect** operation removes duplicates by default.

This example shows that it is possible to write the same query several ways in SQL. This flexibility is beneficial, since it allows a user to think about the query in the way

Telegram: @uni_k

www.konkur.in

that seems most natural. We shall see that there is a substantial amount of redundancy in SQL.

We use the **not in** construct in a way similar to the **in** construct. For example, to find all the courses taught in the Fall 2017 semester but not in the Spring 2018 semester, which we expressed earlier using the **except** operation, we can write:

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id not in (select course_id
                  from section
                  where semester = 'Spring' and year= 2018);
```

The **in** and **not in** operators can also be used on enumerated sets. The following query selects the names of instructors whose names are neither "Mozart" nor "Einstein".

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

In the preceding examples, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. For example, we can write the query "find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011" as follows:

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in (select course_id, sec_id, semester, year
                                              from teaches
                                              where teaches.ID= '10101');
```

Note, however, that some SQL implementations do not support the row construction syntax "(*course_id*, *sec_id*, *semester*, *year*)" used above. We will see alternative ways of writing this query in Section 3.8.3.

3.8.2 Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department." In Section 3.4.1, we wrote this query as follows:

Telegram: @uni_k

www.konkur.in


```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > **some**. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

The subquery:

```
(select salary
 from instructor
 where dept_name = 'Biology')
```

generates the set of all salary values of all instructors in the Biology department. The > **some** comparison in the **where** clause of the outer **select** is true if the *salary* value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and <> **some** comparisons. As an exercise, verify that = **some** is identical to **in**, whereas <> **some** is *not* the same as **not in**.[10]

Now we modify our query slightly. Let us find the names of all instructors that have a salary value greater than that of each instructor in the Biology department. The construct > **all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Biology');
```

As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons. As an exercise, verify that <> **all** is identical to **not in**, whereas = **all** is *not* the same as **in**.

[10]The keyword **any** is synonymous to **some** in SQL. Early versions of SQL allowed only **any**. Later versions added the alternative **some** to avoid the linguistic ambiguity of the word *any* in English.

Telegram: @uni_k

www.konkur.in

As another example of set comparisons, consider the query "Find the departments that have the highest average salary." We begin by writing a query to find all average salaries, and then nest it as a subquery of a larger query that finds those departments for which the average salary is greater than or equal to all average salaries:

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
                            from instructor
                            group by dept_name);
```

3.8.3 Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty. Using the **exists** construct, we can write the query "Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester" in still another way:

```
select course_id
from section as S
where semester = 'Fall' and year= 2017 and
      exists (select *
              from section as T
              where semester = 'Spring' and year= 2018 and
                    S.course_id= T.course_id);
```

The above query also illustrates a feature of SQL where a **correlation name** from an outer query (S in the above query), can be used in a subquery in the **where** clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery**.

In queries that contain subqueries, a scoping rule applies for correlation names. In a subquery, according to the rule, it is legal to use only correlation names defined in the subquery itself or in any query that contains the subquery. If a correlation name is defined both locally in a subquery and globally in a containing query, the local definition applies. This rule is analogous to the usual scoping rules used for variables in programming languages.

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct. We can use the **not exists** construct to simulate the set containment (that is, superset) operation: We can write "relation A contains relation B" as "**not exists** (B **except** A)." (Although it is not part of the current SQL standards, the **contains** operator was present in some early relational systems.) To illustrate the **not exists** operator,

Telegram: @uni_k

www.konkur.in

consider the query "Find all students who have taken all courses offered in the Biology department." Using the **except** construct, we can write the query as follows:

```
select S.ID, S.name
from student as S
where not exists ((select course_id
                   from course
                   where dept_name = 'Biology')
                   except
                   (select T.course_id
                   from takes as T
                   where S.ID = T.ID));
```

Here, the subquery:

```
(select course_id
 from course
 where dept_name = 'Biology')
```

finds the set of all courses offered in the Biology department. The subquery:

```
(select T.course_id
 from takes as T
 where S.ID = T.ID)
```

finds all the courses that student *S.ID* has taken. Thus, the outer **select** takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department.

We saw in Section 3.8.1, an SQL query to "find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011". That query used a tuple constructor syntax that is not supported by some databases. An alternative way to write the query, using the **exists** construct, is as follows:

```
select count (distinct ID)
from takes
where exists (select course_id, sec_id, semester, year
              from teaches
              where teaches.ID= '10101'
                    and takes.course_id = teaches.course_id
                    and takes.sec_id = teaches.sec_id
                    and takes.semester = teaches.semester
                    and takes.year = teaches.year
              );
```

Telegram: @uni_k

www.konkur.in

3.8.4 Test for the Absence of Duplicate Tuples

SQL includes a Boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct[11] returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query "Find all courses that were offered at most once in 2017" as follows:

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id and
                    R.year = 2017);
```

Note that if a course were not offered in 2017, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set.

An equivalent version of this query not using the **unique** construct is:

```
select T.course_id
from course as T
where 1 >= (select count(R.course_id)
            from section as R
            where T.course_id= R.course_id and
                  R.year = 2017);
```

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct. To illustrate this construct, consider the query "Find all courses that were offered at least twice in 2017" as follows:

```
select T.course_id
from course as T
where not unique (select R.course_id
                  from section as R
                  where T.course_id= R.course_id and
                        R.year = 2017);
```

Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two distinct tuples t_1 and t_2 such that $t_1 = t_2$. Since the test $t_1 = t_2$ fails if any of the fields of t_1 or t_2 are null, it is possible for **unique** to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

[11] This construct is not yet widely implemented.

Telegram: @uni_k

www.konkur.in

3.8.5 Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.

Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000." We wrote this query in Section 3.7 by using the **having** clause. We can now rewrite this query, without using the **having** clause, by using a subquery in the **from** clause, as follows:

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

The subquery generates a relation consisting of the names of all departments and their corresponding average instructors' salaries. The attributes of the subquery result can be used in the outer query, as can be seen in the above example.

Note that we do not need to use the **having** clause, since the subquery in the **from** clause computes the average salary, and the predicate that was in the **having** clause earlier is now in the **where** clause of the outer query.

We can give the subquery result relation a name, and rename the attributes, using the **as** clause, as illustrated below.

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
      as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

The subquery result relation is named *dept_avg*, with the attributes *dept_name* and *avg_salary*.

Nested subqueries in the **from** clause are supported by most but not all SQL implementations. Note that some SQL implementations, notably MySQL and PostgreSQL, require that each subquery relation in the **from** clause must be given a name, even if the name is never referenced; Oracle allows a subquery result relation to be given a name (with the keyword **as** omitted) but does not allow renaming of attributes of the relation. An easy workaround for that is to do the attribute renaming in the **select** clause of the subquery; in the above query, the **select** clause of the subquery would be replaced by

```
select dept_name, avg(salary) as avg_salary
```

Telegram: @uni_k

www.konkur.in

and

"**as** *dept_avg* (*dept_name*, *avg_salary*)"

would be replaced by

"**as** *dept_avg*".

As another example, suppose we wish to find the maximum across all departments of the total of all instructors' salaries in each department. The **having** clause does not help us in this task, but we can write this query easily by using a subquery in the **from** clause, as follows:

```
select max (tot_salary)
from (select dept_name, sum(salary)
      from instructor
      group by dept_name) as dept_total (dept_name, tot_salary);
```

We note that nested subqueries in the **from** clause cannot use correlation variables from other relations in the same **from** clause. However, the SQL standard, starting with SQL:2003, allows a subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the same **from** clause. For example, if we wish to print the names of each instructor, along with their salary and the average salary in their department, we could write the query as follows:

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                             from instructor I2
                             where I2.dept_name= I1.dept_name);
```

Without the **lateral** clause, the subquery cannot access the correlation variable *I1* from the outer query. Only the more recent implementations of SQL support the **lateral** clause.

3.8.6 The With Clause

The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which finds those departments with the maximum budget.

```
with max_budget (value) as
      (select max(budget)
      from department)
select budget
from department, max_budget
where department.budget = max budget.value;
```

Telegram: @uni_k

www.konkur.in

The **with** clause in the query defines the temporary relation *max_budget* containing the results of the subquery defining the relation. The relation is available for use only within later parts of the same query. [12] The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.

We could have written the preceding query by using a nested subquery in either the **from** clause or the **where** clause. However, using nested subqueries would have made the query harder to read and understand. The **with** clause makes the query logic clearer; it also permits this temporary relation to be used in multiple places within a query.

For example, suppose we want to find all departments where the total salary is greater than the average of the total salary at all departments. We can write the query using the **with** clause as follows.

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
        from instructor
        group by dept_name),
dept_total_avg(value) as
        (select avg(value)
        from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

We can create an equivalent query without the **with** clause, but it would be more complicated and harder to understand. You can write the equivalent query as an exercise.

3.8.7 Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries**. For example, a subquery can be used in the **select** clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

```
select dept_name,
        (select count(*)
         from instructor
         where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

[12] The SQL evaluation engine may not physically create the relation and is free to compute the overall query result in alternative ways, as long as the result of the query is the same as if the

Telegram: @uni_k

www.konkur.in

The subquery in this example is guaranteed to return only a single value since it has a **count**(*) aggregate without a **group by**. The example also illustrates the usage of correlation variables, that is, attributes of relations in the **from** clause of the outer query, such as *department.dept_name* in the above example.

Scalar subqueries can occur in **select**, **where**, and **having** clauses. Scalar subqueries may also be defined without aggregates. It is not always possible to figure out at compile time if a subquery can return more than one tuple in its result; if the result has more than one tuple when the subquery is executed, a run-time error occurs.

Note that technically the type of a scalar subquery result is still a relation, even if it contains a single tuple. However, when a scalar subquery is used in an expression where a value is expected, SQL implicitly extracts the value from the single attribute of the single tuple in the relation and returns that value.

3.8.8 Scalar Without a From Clause

Certain queries require a calculation but no reference to any relation. Similarly, certain queries may have subqueries that contain a **from** clause without the top-level query needing a **from** clause.

As an example, suppose we wish to find the average number of sections taught (regardless of year or semester) per instructor, with sections taught by multiple instructors counted once per instructor. We need to count the number of tuples in *teaches* to find the total number of sections taught and count the number of tuples in *instructor* to find the number of instructors. Then a simple division gives us the desired result. One might write this as:

(**select count** (*) **from** *teaches*) / (**select count** (*) **from** *instructor*);

While this is legal in some systems, others will report an error due to the lack of a **from** clause.[13] In the latter case, a special dummy relation called, for example, *dual* can be created, containing a single tuple. This allows the preceding query to be written as:

select (**select count** (*) **from** *teaches*) / (**select count** (*) **from** *instructor*)
from *dual*;

Oracle provides a predefined relation called *dual*, containing a single tuple, for uses such as the above (the relation has a single attribute, which is not relevant for our purposes); you can create an equivalent relation if you use any other database.

Since the above queries divide one integer by another, the result would, on most databases, be an integer, which would result in loss of precision. If you wish to get the result as a floating point number, you could multiply one of the two subquery results by 1.0 to convert it to a floating point number, before the division operation is performed.

[13] This construct is legal, for example, in SQL Server, but not legal,

Telegram: @uni_k

www.konkur.in



3.9 Modification of the Database

We have restricted our attention until now to the extraction of information from the database. Now, we show how to add, remove, or change information with SQL.

3.9.1 Deletion

A **delete** request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by:

delete from r
where P;

where P represents a predicate and r represents a relation. The **delete** statement first finds all tuples t in r for which $P(t)$ is true, and then deletes them from r. The **where** clause can be omitted, in which case all tuples in r are deleted.

Telegram: @uni_k

www.konkur.in

Note that a **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation. The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the **where** clause may be empty. The request:

```
delete from instructor;
```

deletes all tuples from the *instructor* relation. The *instructor* relation itself still exists, but it is empty.

Here are examples of SQL delete requests:

- Delete all tuples in the *instructor* relation pertaining to instructors in the Finance department.

```
delete from instructor
where dept_name = 'Finance';
```

- Delete all instructors with a salary between $13,000 and $15,000.

```
delete from instructor
where salary between 13000 and 15000;
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

  This **delete** request first finds all departments located in Watson and then deletes all *instructor* tuples pertaining to those departments.

Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** nested in the **where** clause of a **delete**. The **delete** request can contain a nested **select** that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all instructors with salary below the average at the university. We could write:

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

Telegram: @uni_k

www.konkur.in

The **delete** statement first tests each tuple in the relation *instructor* to check whether the salary is less than the average salary of instructors in the university. Then, all tuples that pass the test—that is, represent an instructor with a lower-than-average salary—are deleted. Performing all the tests before performing any deletion is important—if some tuples are deleted before other tuples have been tested, the average salary may change, and the final result of the **delete** would depend on the order in which the tuples were processed!

3.9.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes.

The simplest **insert** statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title "Database Systems" and four credit hours. We write:

```
insert into course
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

In this example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. For the benefit of users who may not remember the order of the attributes, SQL allows the attributes to be specified as part of the **insert** statement. For example, the following SQL **insert** statements are identical in function to the preceding one:

```
insert into course (course_id, title, dept_name, credits)
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

insert into course (title, course_id, credits, dept_name)
    values ('Database Systems', 'CS-437', 4, 'Comp. Sci.');
```

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000. We write:

```
insert into instructor
    select ID, name, dept_name, 18000
    from student
    where dept_name = 'Music' and tot cred > 144;
```

Telegram: @uni_k

www.konkur.in

Instead of specifying a tuple as we did earlier in this section, we use a **select** to specify a set of tuples. SQL evaluates the **select** statement first, giving a set of tuples that is then inserted into the *instructor* relation. Each tuple has an *ID*, a *name*, a *dept_name* (Music), and a salary of $18,000.

It is important that the system evaluate the **select** statement fully before it performs any insertions. If it were to carry out some insertions while the **select** statement was being evaluated, a request such as:

```
insert into student
    select *
    from student;
```

might insert an infinite number of tuples, if the primary key constraint on *student* were absent. Without the primary key constraint, the request would insert the first tuple in *student* again, creating a second copy of the tuple. Since this second copy is part of *student* now, the **select** statement may find it, and a third copy would be inserted into *student*. The **select** statement may then find this third copy and insert a fourth copy, and so on, forever. Evaluating the **select** statement completely before performing insertions avoids such problems. Thus, the above **insert** statement would simply duplicate every tuple in the *student* relation if the relation did not have a primary key constraint.

Our discussion of the **insert** statement considered only examples in which a value is given for every attribute in inserted tuples. It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by *null*. Consider the request:

```
insert into student
    values ('3003', 'Green', 'Finance', null);
```

The tuple inserted by this request specified that a student with *ID* "3003" is in the Finance department, but the *tot_cred* value for this student is not known.

Most relational database products have special "bulk loader" utilities to insert a large set of tuples into a relation. These utilities allow data to be read from formatted text files, and they can execute much faster than an equivalent sequence of insert statements.

3.9.3 Updates

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used. As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query.

Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent. We write:

Telegram: @uni_k

www.konkur.in


```
update instructor
set salary= salary * 1.05;
```

The preceding update statement is applied once to each of the tuples in the *instructor* relation.

If a salary increase is to be paid only to instructors with a salary of less than $70,000, we can write:

```
update instructor
set salary = salary * 1.05
where salary < 70000;
```

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **select**s). As with **insert** and **delete**, a nested **select** within an **update** statement may reference the relation that is being updated. As before, SQL first tests all tuples in the relation to see whether they should be updated, and it carries out the updates afterward. For example, we can write the request "Give a 5 percent salary raise to instructors whose salary is less than average" as follows:

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

Let us now suppose that all instructors with salary over $100,000 receive a 3 percent raise, whereas all others receive a 5 percent raise. We could write two **update** statements:

```
update instructor
set salary = salary * 1.03
where salary > 100000;

update instructor
set salary = salary * 1.05
where salary <= 100000;
```

Note that the order of the two **update** statements is important. If we changed the order of the two statements, an instructor with a salary just under $100,000 would receive a raise of over 8 percent.

SQL provides a **case** construct that we can use to perform both updates with a single **update** statement, avoiding the problem with the order of updates.

Telegram: @uni_k

www.konkur.in


```
update instructor
set salary = case
                when salary <= 100000 then salary * 1.05
                else salary * 1.03
             end
```

The general form of the case statement is as follows:

```
case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end
```

The operation returns $result_i$, where i is the first of $pred_1, pred_2, \ldots, pred_n$ that is satisfied; if none of the predicates is satisfied, the operation returns $result_0$. Case statements can be used in any place where a value is expected.

Scalar subqueries are useful in SQL update statements, where they can be used in the **set** clause. We illustrate this using the *student* and *takes* relations that we introduced in Chapter 2. Consider an update where we set the *tot_cred* attribute of each *student* tuple to the sum of the credits of courses successfully completed by the student. We assume that a course is successfully completed if the student has a grade that is neither 'F' nor null. To specify this update, we need to use a subquery in the **set** clause, as shown below:

```
update student
set tot_cred = (
    select sum(credits)
    from takes, course
    where student.ID= takes.ID and
                takes.course_id = course.course_id and
                takes.grade <> 'F' and
                takes.grade is not null);
```

In case a student has not successfully completed any course, the preceding statement would set the *tot_cred* attribute value to null. To set the value to 0 instead, we could use another **update** statement to replace null values with 0; a better alternative is to replace the clause "**select sum**(*credits*)" in the preceding subquery with the following **select** clause using a **case** expression:

```
select case
        when sum(credits) is not null then sum(credits)
        else 0
        end
```

Telegram: @uni_k

www.konkur.in

Many systems support a **coalesce** function, which we describe in more detail later, in Section 4.5.2, which provides a concise way of replacing nulls by other values. In the above example, we could have used **coalesce**(**sum**(*credits*), 0) instead of the **case** expression; this expression would return the aggregate result **sum**(*credits*) if it is not null, and 0 otherwise.

3.10 Summary

- SQL is the most influential commercially marketed relational query language. The SQL language has several parts:
  - **Data-definition language** (DDL), which provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
  - **Data-manipulation language** (DML), which includes a query language and commands to insert tuples into, delete tuples from, and modify tuples in the database.
- The SQL data-definition language is used to create relations with specified schemas. In addition to specifying the names and types of relation attributes, SQL also allows the specification of integrity constraints such as primary-key constraints and foreign-key constraints.
- SQL includes a variety of language constructs for queries on the database. These include the **select**, **from**, and **where** clauses.
- SQL also provides mechanisms to rename both attributes and relations, and to order query results by sorting on specified attributes.
- SQL supports basic set operations on relations, including **union**, **intersect**, and **except**, which correspond to the mathematical set operations $\cup$, $\cap$, and $-$.
- SQL handles queries on relations containing null values by adding the truth value "unknown" to the usual truth values of true and false.
- SQL supports aggregation, including the ability to divide a relation into groups, applying aggregation separately on each group. SQL also supports set operations on groups.
- SQL supports nested subqueries in the **where** and **from** clauses of an outer query. It also supports scalar subqueries wherever an expression returning a value is permitted.
- SQL provides constructs for updating, inserting, and deleting information.

Telegram: @uni_k

www.konkur.in

Review Terms

- Data-definition language
- Data-manipulation language
- Database schema
- Database instance
- Relation schema
- Relation instance
- Primary key
- Foreign key
  - Referencing relation
  - Referenced relation
- Null value
- Query language
- SQL query structure
  - **select** clause
  - **from** clause
  - **where** clause
- Multiset relational algebra
- **as** clause
- **order by** clause
- Table alias
- Correlation name (correlation variable, tuple variable)
- Set operations
  - **union**
  - **intersect**
  - **except**
- Aggregate functions
  - **avg, min, max, sum, count**
  - **group by**
  - **having**
- Nested subqueries
- Set comparisons
  - {<, <=, >, >=} { **some, all** }
  - **exists**
  - **unique**
- **lateral** clause
- **with** clause
- Scalar subquery
- Database modification
  - Delete
  - Insert
  - Update

Practice Exercises

3.1 Write the following queries in SQL, using the university schema. (We suggest you actually run these queries on a database, using the sample data that we provide on the web site of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above web site.)

a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

Telegram: @uni_k

www.konkur.in

c. Find the highest salary of any instructor.

d. Find all instructors earning the highest salary (there may be more than one with the same salary).

e. Find the enrollment of each section that was offered in Fall 2017.

f. Find the maximum enrollment, across all sections, in Fall 2017.

g. Find the sections that had the maximum enrollment in Fall 2017.

3.2 Suppose you are given a relation *grade_points*(*grade*, *points*) that provides a conversion from letter grades in the *takes* relation to numeric scores; for example, an "A" grade could be specified to correspond to 4 points, an "A−" to 3.7 points, a "B+" to 3.3 points, a "B" to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received.

Given the preceding relation, and our university schema, write each of the following queries in SQL. You may assume for simplicity that no *takes* tuple has the *null* value for *grade*.

a. Find the total grade points earned by the student with ID '12345', across all courses taken by the student.

b. Find the grade point average (*GPA*) for the above student, that is, the total grade points divided by the total credits for the associated courses.

c. Find the ID and the grade-point average of each student.

d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly.

3.3 Write the following inserts, deletes, or updates in SQL, using the university schema.

a. Increase the salary of each instructor in the Comp. Sci. department by 10%.

b. Delete all courses that have never been offered (i.e., do not occur in the *section* relation).

c. Insert every student whose *tot_cred* attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.

3.4 Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the total number of people who owned cars that were involved in accidents in 2017.

Telegram: @uni_k

www.konkur.in

person (*driver_id*, *name*, *address*)
car (*license_plate*, *model*, *year*)
accident (*report_number*, *year*, *location*)
owns (*driver_id*, *license_plate*)
participated (*report_number*, *license_plate*, *driver_id*, *damage_amount*)

Figure 3.17 Insurance database

b. Delete all year-2010 cars belonging to the person whose ID is '12345'.

3.5 Suppose that we have a relation *marks*(*ID*, *score*) and we wish to assign grades to students based on the score as follows: grade *F* if *score* < 40, grade *C* if 40 ≤ *score* < 60, grade *B* if 60 ≤ *score* < 80, and grade *A* if 80 ≤ *score*. Write SQL queries to do the following:

a. Display the grade for each student, based on the *marks* relation.

b. Find the number of students with each grade.

3.6 The SQL **like** operator is case sensitive (in most systems), but the lower() function on strings can be used to perform case-insensitive matching. To show how, write a query that finds departments whose names contain the string "sci" as a substring, regardless of the case.

3.7 Consider the SQL query

select $p.a1$
from p, $r1$, $r2$
where $p.a1 = r1.a1$ **or** $p.a1 = r2.a1$

Under what conditions does the preceding query select values of $p.a1$ that are either in $r1$ or in $r2$? Examine carefully the cases where either $r1$ or $r2$ may be empty.

3.8 Consider the bank database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the ID of each customer of the bank who has an account but not a loan.

b. Find the ID of each customer who lives on the same street and in the same city as customer '12345'.

c. Find the name of each branch that has at least one customer who has an account in the bank and who lives in "Harrison".

Telegram: @uni_k

www.konkur.in

branch(*branch_name*, *branch_city, assets*)
customer (*ID*, *customer_name*, *customer_street, customer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*ID*, *loan_number*)
account (*account_number*, *branch_name, balance*)
depositor (*ID*, *account_number*)

Figure 3.18 Banking database.

3.9 Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the ID, name, and city of residence of each employee who works for "First Bank Corporation".

b. Find the ID, name, and city of residence of each employee who works for "First Bank Corporation" and earns more than $10000.

c. Find the ID of each employee who does not work for "First Bank Corporation".

d. Find the ID of each employee who earns more than every employee of "Small Bank Corporation".

e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which "Small Bank Corporation" is located.

f. Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).

g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 3.19 Employee database.

Telegram: @uni_k

www.konkur.in

3.10 Consider the relational database of Figure 3.19. Give an expression in SQL for each of the following:

a. Modify the database so that the employee whose ID is '12345' now lives in "Newtown".

b. Give each manager of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than $100000; in such cases, give only a 3 percent raise.

Exercises

3.11 Write the following queries in SQL, using the university schema.

a. Find the ID and name of each student who has taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

b. Find the ID and name of each student who has not taken any course offered before 2017.

c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

3.12 Write the SQL statements using the university schema to perform the following operations:

a. Create a new course "CS-001", titled "Weekly Seminar", with 0 credits.

b. Create a section of this course in Fall 2017, with *sec_id* of 1, and with the location of this section not yet specified.

c. Enroll every student in the Comp. Sci. department in the above section.

d. Delete enrollments in the above section where the student's ID is 12345.

e. Delete the course CS-001. What will happen if you run this **delete** statement without first deleting offerings (sections) of this course?

f. Delete all *takes* tuples corresponding to any section of any course with the word "advanced" as a part of the title; ignore case when matching the word with the title.

3.13 Write SQL DDL corresponding to the schema in Figure 3.17. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

Telegram: @uni_k

www.konkur.in

3.14 Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the number of accidents involving a car belonging to a person named "John Smith".

b. Update the damage amount for the car with license_plate "AABB2000" in the accident with report number "AR2197" to $3000.

3.15 Consider the bank database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find each customer who has an account at *every* branch located in "Brooklyn".

b. Find the total sum of all loan amounts in the bank.

c. Find the names of all branches that have assets greater than those of at least one branch located in "Brooklyn".

3.16 Consider the employee database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.

b. Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.

c. Find ID and name of each employee who earns more than the average salary of all employees of her or his company.

d. Find the company that has the smallest payroll.

3.17 Consider the employee database of Figure 3.19. Give an expression in SQL for each of the following queries.

a. Give all employees of "First Bank Corporation" a 10 percent raise.

b. Give all managers of "First Bank Corporation" a 10 percent raise.

c. Delete all tuples in the *works* relation for employees of "Small Bank Corporation".

3.18 Give an SQL schema definition for the employee database of Figure 3.19. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema. Include any foreign-key constraints that might be appropriate.

3.19 List two reasons why null values might be introduced into the database.

3.20 Show that, in SQL, <> **all** is identical to **not in**.

Telegram: @uni_k

www.konkur.in

member(*memb_no*, *name*)
book(*isbn*, *title*, *authors*, *publisher*)
borrowed(*memb_no*, *isbn*, *date*)

Figure 3.20 Library database.

3.21 Consider the library database of Figure 3.20. Write the following queries in SQL.

a. Find the member number and name of each member who has borrowed at least one book published by "McGraw-Hill".

b. Find the member number and name of each member who has borrowed every book published by "McGraw-Hill".

c. For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.

d. Find the average number of books borrowed per member. Take into account that if a member does not borrow any books, then that member does not appear in the *borrowed* relation at all, but that member still counts in the average.

3.22 Rewrite the **where** clause

where unique (**select** *title* **from** *course*)

without using the **unique** construct.

3.23 Consider the query:

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
        from instructor
        group by dept_name),
dept_total_avg(value) as
        (select avg(value)
        from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Rewrite this query without using the **with** construct.

3.24 Using the university schema, write an SQL query to find the name and ID of those Accounting students advised by an instructor in the Physics department.

Telegram: @uni_k

www.konkur.in

3.25 Using the university schema, write an SQL query to find the names of those departments whose budget is higher than that of Philosophy. List them in alphabetic order.

3.26 Using the university schema, use SQL to do the following: For each student who has retaken a course at least twice (i.e., the student has taken the course at least three times), show the course ID and the student's ID.
Please display your results in order of course ID and do not display duplicate rows.

3.27 Using the university schema, write an SQL query to find the IDs of those students who have retaken at least three distinct courses at least once (i.e, the student has taken the course at least two times).

3.28 Using the university schema, write an SQL query to find the names and IDs of those instructors who teach every course taught in his or her department (i.e., every course that appears in the *course* relation with the instructor's department name). Order result by name.

3.29 Using the university schema, write an SQL query to find the name and ID of each History student whose name begins with the letter 'D' and who has *not* taken at least five Music courses.

3.30 Consider the following SQL query on the university schema:

select avg(*salary*) - (**sum**(*salary*) / **count**(*))
from *instructor*

We might expect that the result of this query is zero since the average of a set of numbers is defined to be the sum of the numbers divided by the number of numbers. Indeed this is true for the example *instructor* relation in Figure 2.1. However, there are other possible instances of that relation for which the result would *not* be zero. Give one such instance, and explain why the result would not be zero.

3.31 Using the university schema, write an SQL query to find the ID and name of each instructor who has never given an A grade in any course she or he has taught. (Instructors who have never taught a course trivially satisfy this condition.)

3.32 Rewrite the preceding query, but also ensure that you include only instructors who have given at least one other non-null grade in some course.

3.33 Using the university schema, write an SQL query to find the ID and title of each course in Comp. Sci. that has had at least one section with afternoon hours (i.e., ends at or after 12:00). (You should eliminate duplicates if any.)

3.34 Using the university schema, write an SQL query to find the number of students in each section. The result columns should appear in the order "courseid, secid, year, semester, num". You do not need to output sections with 0 students.

Telegram: @uni_k

www.konkur.in

3.35 Using the university schema, write an SQL query to find section(s) with maximum enrollment. The result columns should appear in the order "courseid, secid, year, semester, num". (It may be convenient to use the *with* construct.)

Tools

A number of relational database systems are available commercially, including IBM DB2, IBM Informix, Oracle, SAP Adaptive Server Enterprise (formerly Sybase), and Microsoft SQL Server. In addition several open-source database systems can be downloaded and used free of charge, including PostgreSQL and MySQL (free except for certain kinds of commercial use). Some commercial vendors offer free versions of their systems with certain use limitations. These include Oracle Express edition, Microsoft SQL Server Express, and IBM DB2 Express-C.

The sql.js database is version of the embedded SQL database SQLite which can be run directly in a web browser, allowing SQL commands to be executed directly in the browser. All data are temporary and vanishes when you close the browser, but it can be useful for learning SQL; be warned that the subset of SQL that is supported by sql.js and SQLite is considerably smaller than what is supported by other databases. An SQL tutorial using sql.js as the execution engine is hosted at www.w3schools.com/sql.

The web site of our book, db-book.com, provides a significant amount of supporting material for the book. By following the link on the site titled Laboratory Material, you can get access to the following:

- Instructions on how to set up and access some popular database systems, including sql.js (which you can run in your browser), MySQL, and PostgreSQL.
- SQL schema definitions for the University schema.
- SQL scripts for loading sample datasets.
- Tips on how to use the XData system, developed at IIT Bombay, to test queries for correctness by executing them on multiple datasets generated by the system; and, for instructors, tips on how to use XData to automate SQL query grading.
- Get tips on SQL variations across different databases.

Support for different SQL features varies by databases, and most databases also support some non-standard extensions to SQL. Read the system manuals to understand the exact SQL features that a database supports.

Most database systems provide a command line interface for submitting SQL commands. In addition, most databases also provide graphical user interfaces (GUIs), which simplify the task of browsing the database, creating and submitting queries, and administering the database. For PostgreSQL, the pgAdmin tool provides GUI functionality, while for MySQL, phpMyAdmin provides functionality. Oracle provides

Telegram: @uni_k

www.konkur.in

Oracle SQL Developer, while Microsoft SQL Server comes with the SQL Server Management Studio.

The NetBeans IDEs SQLEditor provides a GUI front end which works with a number of different database systems, but with limited functionality, while the Eclipse IDE supports similar functionality through the Data Tools Platform (DTP). Commercial IDEs that support SQL access across multiple database platforms include Embarcadero's RAD Studio and Aqua Data Studio.

Further Reading

The original Sequel language that became SQL is described in [Chamberlin et al. (1976)].

The most important SQL reference is likely to be the online documentation provided by the vendor or the particular database system you are using. That documentation will identify any features that deviate from the SQL standard features presented in this chapter. Here are links to the SQL reference manuals for the current (as of 2018) versions of some of the popular databases.

- MySQL 8.0: dev.mysql.com/doc/refman/8.0/en/
- Oracle 12c: docs.oracle.com/database/121/SQLRF/
- PostgreSQL: www.postgresql.org/docs/current/static/sql.html
- SQLite: www.sqlite.org/lang.html
- SQL Server: docs.microsoft.com/en-us/sql/t-sql

Bibliography

[Chamberlin et al. (1976)] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development*, Volume 20, Number 6 (1976), pages 560–575.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 4



Intermediate SQL

In this chapter, we continue our study of SQL. We consider more complex forms of SQL queries, view definition, transactions, integrity constraints, more details regarding SQL data definition, and authorization.

4.1 Join Expressions

In all of the example queries we used in Chapter 3 (except when we used set operations), we combined information from multiple relations using the Cartesian product operator. In this section, we introduce a number of "join" operations that allow the programmer to write some queries in a more natural way and to express some queries that are difficult to do with only the Cartesian product.

00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.1 The *student* relation.

Telegram: @uni_k

www.konkur.in

00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

Figure 4.2 The *takes* relation.

All the examples used in this section involve the two relations *student* and *takes*, shown in Figure 4.1 and Figure 4.2, respectively. Observe that the attribute *grade* has a value null for the student with *ID* 98988, for the course BIO-301, section 1, taken in Summer 2018. The null value indicates that the grade has not been awarded yet.

4.1.1 The Natural Join

Consider the following SQL query, which computes for each student the set of courses a student has taken:

select *name*, *course_id*
from *student*, *takes*
where *student.ID* = *takes.ID*;

Note that this query outputs only students who have taken some course. Students who have not taken any course are not output.

Telegram: @uni_k

www.konkur.in

Note that in the *student* and *takes* table, the matching condition required *student.ID* to be equal to *takes.ID*. These are the only attributes in the two relations that have the same name. In fact, this is a common case; that is, the matching condition in the **from** clause most often requires all attributes with matching names to be equated.

To make the life of an SQL programmer easier for this common case, SQL supports an operation called the *natural join*, which we describe below. In fact, SQL supports several other ways in which information from two or more relations can be **joined** together. We have already seen how a Cartesian product along with a **where** clause predicate can be used to join information from multiple relations. Other ways of joining information from multiple relations are discussed in Section 4.1.2 through Section 4.1.4.

The **natural join** operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations. So, going back to the example of the relations *student* and *takes*, computing:

student **natural join** *takes*

considers only those pairs of tuples where both the tuple from *student* and the tuple from *takes* have the same value on the common attribute, *ID*.

The resulting relation, shown in Figure 4.3, has only 22 tuples, the ones that give information about a student and a course that the student has actually taken. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once. Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

Earlier we wrote the query "For all students in the university who have taken some course, find their names and the course ID of all courses they took" as:

select *name*, *course_id*
from *student*, *takes*
where *student.ID* = *takes.ID*;

This query can be written more concisely using the natural-join operation in SQL as:

select *name*, *course_id*
from *student* **natural join** *takes*;

Both of the above queries generate the same result.[1]

[1] For notational symmetry, SQL allows the Cartesian product, which we have denoted with a comma, to be denoted by the keywords **cross join**. Thus, "**from** *student*, *takes*" could be

Telegram: @uni_k

www.konkur.in

00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Figure 4.3 The natural join of the *student* relation with the *takes* relation.

The result of the natural join operation is a relation. Conceptually, expression "*student* **natural join** *takes*" in the **from** clause is replaced by the relation obtained by evaluating the natural join.[2] The **where** and **select** clauses are then evaluated on this relation, as we saw in Section 3.3.2.

A **from** clause in an SQL query can have multiple relations combined using natural join, as shown here:

select $A_1, A_2, \ldots, A_n$
from r_1 **natural join** r_2 **natural join** $\ldots$ **natural join** r_m
where P;

More generally, a **from** clause can be of the form

[2]As a consequence, it may not be possible in some systems to use attribute names containing the original relation names, for instance, *student.ID* or *takes.ID*, to refer to attributes in the natural join result. While some systems allow it, others don't, and some allow it for all attributes except the join attributes (i.e., those that appear in both relation schemas). We can, however, use attribute names such as *name* and *course_id*

Telegram: @uni_k

www.konkur.in

from $E_1, E_2, \ldots, E_n$

where each E_i can be a single relation or an expression involving natural joins. For example, suppose we wish to answer the query "List the names of students along with the titles of courses that they have taken." The query can be written in SQL as follows:

select *name*, *title*
from *student* **natural join** *takes*, *course*
where *takes.course_id* = *course.course_id*;

The natural join of *student* and *takes* is first computed, as we saw earlier, and a Cartesian product of this result with *course* is computed, from which the **where** clause extracts only those tuples where the course identifier from the join result matches the course identifier from the *course* relation. Note that *takes.course_id* in the **where** clause refers to the *course_id* field of the natural join result, since this field, in turn, came from the *takes* relation.

In contrast, the following SQL query does *not* compute the same result:

select *name*, *title*
from *student* **natural join** *takes* **natural join** *course*;

To see why, note that the natural join of *student* and *takes* contains the attributes (*ID*, *name*, *dept_name*, *tot_cred*, *course_id*, *sec_id*), while the *course* relation contains the attributes (*course_id*, *title*, *dept_name*, *credits*). As a result, the natural join would require that the *dept_name* attribute values from the two relations be the same in addition to requiring that the *course_id* values be the same. This query would then omit all (student name, course title) pairs where the student takes a course in a department other than the student's own department. The previous query, on the other hand, correctly outputs such pairs.

To provide the benefit of natural join while avoiding the danger of equating attributes erroneously, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated. This feature is illustrated by the following query:

select *name*, *title*
from (*student* **natural join** *takes*) **join** *course* **using** (*course_id*);

The operation **join** … **using** requires a list of attribute names to be specified. Both relations being joined must have attributes with the specified names. Consider the operation r_1 **join** r_2 **using**(A_1, A_2). The operation is similar to r_1 **natural join** r_2, except that a pair of tuples t_1 from r_1 and t_2 from r_2 match if $t_1.A_1 = t_2.A_1$ and $t_1.A_2 = t_2.A_2$; even if r_1 and r_2 both have an attribute named A_3, it is *not* required that $t.A = t.A$.

Telegram: @uni_k

www.konkur.in

Thus, in the preceding SQL query, the **join** construct permits *student.dept_name* and *course.dept_name* to differ, and the SQL query gives the correct answer.

4.1.2 Join Conditions

In Section 4.1.1, we saw how to express natural joins, and we saw the **join** … **using** clause, which is a form of natural join that requires values to match only on specified attributes. SQL supports another form of join, in which an arbitrary join condition can be specified.

The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression.

Consider the following query, which has a join expression containing the **on** condition:

select *
from *student* **join** *takes* **on** *student.ID* = *takes.ID*;

The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal. The join expression in this case is almost the same as the join expression *student* **natural join** *takes*, since the natural join operation also requires that for a *student* tuple and a *takes* tuple to match. The one difference is that the result has the *ID* attribute listed twice, in the join result, once for *student* and once for *takes*, even though their *ID* values must be the same.

In fact, the preceding query is equivalent to the following query:

select *
from *student*, *takes*
where *student.ID* = *takes.ID*;

As we have seen earlier, the relation name is used to disambiguate the attribute name *ID*, and thus the two occurrences can be referred to as *student.ID* and *takes.ID*, respectively. A version of this query that displays the *ID* value only once is as follows:

select *student.ID* **as** *ID*, *name*, *dept_name*, *tot_cred*,
course_id, *sec_id*, *semester*, *year*, *grade*
from *student* **join** *takes* **on** *student.ID* = *takes.ID*;

The result of this query is exactly the same as the result of the natural join of *student* and *takes*, which we showed in Figure 4.3.

The **on** condition can express any SQL predicate, and thus join expressions using the **on** condition can express a richer class of join conditions than **natural join**. However,

Telegram: @uni_k

www.konkur.in

as illustrated by our preceding example, a query using a join expression with an **on** condition can be replaced by an equivalent expression without the **on** condition, with the predicate in the **on** clause moved to the **where** clause. Thus, it may appear that the **on** condition is a redundant feature of SQL.

However, there are two good reasons for introducing the **on** condition. First, we shall see shortly that for a kind of join called an outer join, **on** conditions do behave in a manner different from **where** conditions. Second, an SQL query is often more readable by humans if the join condition is specified in the **on** clause and the rest of the conditions appear in the **where** clause.

4.1.3 Outer Joins

Suppose we wish to display a list of all students, displaying their *ID*, and *name*, *dept_name*, and *tot_cred*, along with the courses that they have taken. The following SQL query may appear to retrieve the required information:

```
select *
from student natural join takes;
```

Unfortunately, the above query does not work quite as intended. Suppose that there is some student who takes no courses. Then the tuple in the *student* relation for that particular student would not satisfy the condition of a natural join with any tuple in the *takes* relation, and that student's data would not appear in the result. We would thus not see any information about students who have not taken any courses. For example, in the *student* and *takes* relations of Figure 4.1 and Figure 4.2, note that student Snow, with ID 70557, has not taken any courses. Snow appears in *student*, but Snow's ID number does not appear in the *ID* column of *takes*. Thus, Snow does not appear in the result of the natural join.

More generally, some tuples in either or both of the relations being joined may be "lost" in this way. The **outer-join** operation works in a manner similar to the join operations we have already studied, but it preserves those tuples that would be lost in a join by creating tuples in the result containing null values.

For example, to ensure that the student named Snow from our earlier example appears in the result, a tuple could be added to the join result with all attributes from the *student* relation set to the corresponding values for the student Snow, and all the remaining attributes which come from the *takes* relation, namely, *course_id*, *sec_id*, *semester*, and *year*, set to *null*. Thus, the tuple for the student Snow is preserved in the result of the outer join.

There are three forms of outer join:

- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.

Telegram: @uni_k

www.konkur.in

- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve nonmatched tuples are called **inner-join** operations, to distinguish them from the outer-join operations.

We now explain exactly how each form of outer join operates. We can compute the left outer-join operation as follows: First, compute the result of the inner join as before. Then, for every tuple t in the left-hand-side relation that does not match any tuple in the right-hand-side relation in the inner join, add a tuple r to the result of the join constructed as follows:

- The attributes of tuple r that are derived from the left-hand-side relation are filled in with the values from tuple t.
- The remaining attributes of r are filled with null values.

Figure 4.4 shows the result of:

```
select *
from student natural left outer join takes;
```

That result includes student Snow (*ID* 70557), unlike the result of an inner join, but the tuple for Snow includes nulls for the attributes that appear only in the schema of the *takes* relation.[3]

As another example of the use of the outer-join operation, we can write the query "Find all students who have not taken a course" as:

```
select ID
from student natural left outer join takes
where course_id is null;
```

The **right outer join** is symmetric to the **left outer join**. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. Thus, if we rewrite the preceding query using a right outer join and swapping the order in which we list the relations as follows:

```
select *
from takes natural right outer join student;
```

we get the same result except for the order in which the attributes appear in the result (see Figure 4.5).

[3] We show null values in tables using *null*, but most systems display null

Telegram: @uni_k

www.konkur.in

00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Figure 4.4 Result of *student* **natural left outer join** *takes*.

The **full outer join** is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls those tuples from the left-hand-side relation that did not match with any from the right-hand-side relation and adds them to the result. Similarly, it extends with nulls those tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result. Said differently, full outer join is the union of a left outer join and the corresponding right outer join.[4]

As an example of the use of full outer join, consider the following query: "Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2017; all course sections from Spring 2017 must

[4]In those systems, notably MySQL, that implement only left and right outer join, this is exactly how one has to write a full outer join.

Telegram: @uni_k

www.konkur.in

00128	CS-101	1	Fall	2017	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2017	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2017	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2017	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2018	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2017	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2018	B	Brandt	History	80
23121	FIN-201	1	Spring	2018	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2017	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2017	F	Levy	Physics	46
45678	CS-101	1	Spring	2018	B+	Levy	Physics	46
45678	CS-319	1	Spring	2018	B	Levy	Physics	46
54321	CS-101	1	Fall	2017	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2017	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2018	A-	Sanchez	Music	38
70557	*null*	*null*	*null*	*null*	*null*	Snow	Physics	0
76543	CS-101	1	Fall	2017	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2018	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2017	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2017	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2018	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2017	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2018	*null*	Tanaka	Biology	120

Figure 4.5 The result of *takes* **natural right outer join** *student*.

be displayed, even if no student from the Comp. Sci. department has taken the course section." This query can be written as:

```
select *
from (select *
      from student
      where dept_name = 'Comp. Sci.')
     natural full outer join
     (select *
      from takes
      where semester = 'Spring' and year = 2017);
```

The result appears in Figure 4.6.

Telegram: @uni_k

www.konkur.in

00128	Zhang	Comp. Sci.	102	*null*	*null*	*null*	*null*	*null*
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
76543	Brown	Comp. Sci.	58	*null*	*null*	*null*	*null*	*null*
76653	*null*	*null*	*null*	ECE-181	1	Spring	2017	C

Figure 4.6 Result of full outer join example (see text).

The **on** clause can be used with outer joins. The following query is identical to the first query we saw using "*student* **natural left outer join** *takes*," except that the attribute *ID* appears twice in the result.

```
select *
from student left outer join takes on student.ID = takes.ID;
```

As we noted earlier, **on** and **where** behave differently for outer join. The reason for this is that outer join adds null-padded tuples only for those tuples that do not contribute to the result of the corresponding "inner" join. The **on** condition is part of the outer join specification, but a **where** clause is not. In our example, the case of the *student* tuple for student "Snow" with ID 70557, illustrates this distinction. Suppose we modify the preceding query by moving the **on** clause predicate to the **where** clause and instead using an **on** condition of *true*.[5]

```
select *
from student left outer join takes on true
where student.ID = takes.ID;
```

The earlier query, using the left outer join with the **on** condition, includes a tuple (70557, Snow, Physics, 0, *null*, *null*, *null*, *null*, *null*, *null*) because there is no tuple in *takes* with *ID* = 70557. In the latter query, however, every tuple satisfies the join condition *true*, so no null-padded tuples are generated by the outer join. The outer join actually generates the Cartesian product of the two relations. Since there is no tuple in *takes* with *ID* = 70557, every time a tuple appears in the outer join with *name* = "Snow", the values for *student.ID* and *takes.ID* must be different, and such tuples would be eliminated by the **where** clause predicate. Thus, student Snow never appears in the result of the latter query.

[5] Some systems do not allow the use of the Boolean constant *true*. To test this on those systems, use a tautology (i.e., a predicate that always evaluates to true), like "1=1".

Telegram: @uni_k

www.konkur.in



4.1.4 Join Types and Conditions

To distinguish normal joins from outer joins, normal joins are called **inner joins** in SQL. A join clause can thus specify **inner join** instead of **outer join** to specify that a normal join is to be used. The keyword **inner** is, however, optional. The default join type, when the **join** clause is used without the **outer** prefix, is the **inner join**. Thus,

```
select *
from student join takes using (ID);
```

is equivalent to:

```
select *
from student inner join takes using (ID);
```

Similarly, **natural join** is equivalent to **natural inner join**.

Figure 4.7 shows a full list of the various types of join that we have discussed. As can be seen from the figure, any form of join (inner, left outer, right outer, or full outer) can be combined with any join condition (natural, using, or on).

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on < predicate>
using $(A_1, A_2, \ldots, A_n)$

Figure 4.7 Join types and join conditions.

Telegram: @uni_k

www.konkur.in

4.2 Views

It is not always desirable for all users to see the entire set of relations in the database. In Section 4.7, we shall see how to use the SQL authorization mechanism to restrict access to relations, but security considerations may require that only certain data in a relation be hidden from a user. Consider a clerk who needs to know an instructor's ID, name, and department name, but does not have authorization to see the instructor's salary amount. This person should see a relation described in SQL by:

```
select ID, name, dept_name
from instructor;
```

Aside from security concerns, we may wish to create a personalized collection of "virtual" relations that is better matched to a certain user's intuition of the structure of the enterprise. In our university example, we may want to have a list of all course sections offered by the Physics department in the Fall 2017 semester, with the building and room number of each section. The relation that we would create for obtaining such a list is:

```
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
      and course.dept_name = 'Physics'
      and section.semester = 'Fall'
      and section.year = 2017;
```

It is possible to compute and store the results of these queries and then make the stored relations available to users. However, if we did so, and the underlying data in the relations *instructor*, *course*, or *section* changed, the stored query results would then no longer match the result of reexecuting the query on the relations. In general, it is a bad idea to compute and store query results such as those in the above examples (although there are some exceptions that we study later).

Instead, SQL allows a "virtual relation" to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored but instead is computed by executing the query whenever the virtual relation is used. We saw a feature for this in Section 3.8.6, where we described the **with** clause. The **with** clause allows us to to assign a name to a subquery for use as often as desired, but in one particular query only. Here, we present a way to extend this concept beyond a single query by defining a **view**. It is possible to support a large number of views on top of any given set of actual relations.

Telegram: @uni_k

www.konkur.in

4.2.1 View Definition

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is:

create view *v* **as** <query expression>;

where <query expression> is any legal query expression. The view name is represented by *v*.

Consider again the clerk who needs to access all data in the *instructor* relation, except *salary*. The clerk should not be authorized to access the *instructor* relation (we see in Section 4.7, how authorizations can be specified). Instead, a view relation *faculty* can be made available to the clerk, with the view defined as follows:

create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*;

As explained earlier, the view relation conceptually contains the tuples in the query result, but it is not precomputed and stored. Instead, the database system stores the query expression associated with the view relation. Whenever the view relation is accessed, its tuples are created by computing the query result. Thus, the view relation is created whenever needed, on demand.

To create a view that lists all course sections offered by the Physics department in the Fall 2017 semester with the building and room number of each section, we write:

create view *physics_fall_2017* **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
    and *course.dept_name* = 'Physics'
    and *section.semester* = 'Fall'
    and *section.year* = 2017;

Later, when we study the SQL authorization mechanism in Section 4.7, we shall see that users can be given access to views in place of, or in addition to, access to relations.

Views differ from the **with** statement in that views, once created, remain available until explicitly dropped. The named subquery defined by **with** is local to the query in which it is defined.

4.2.2 Using Views in SQL Queries

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view *physics_fall_2017*, we can find all Physics courses offered in the Fall 2017 semester in the Watson building

Telegram: @uni_k

www.konkur.in


```
select course_id
from physics_fall_2017
where building = 'Watson';
```

View names may appear in a query any place where a relation name may appear,

The attribute names of a view can be specified explicitly as follows:

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

The preceding view gives for each department the sum of the salaries of all the instructors at that department. Since the expression **sum**(*salary*) does not have a name, the attribute name is specified explicitly in the view definition.

Intuitively, at any given time, the set of tuples in the view relation is the result of evaluation of the query expression that defines the view. Thus, if a view relation is computed and stored, it may become out of date if the relations used to define it are modified. To avoid this, views are usually implemented as follows: When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the query expression that defines the view. Wherever a view relation appears in a query, it is replaced by the stored query expression. Thus, whenever we evaluate the query, the view relation is recomputed.

One view may be used in the expression defining another view. For example, we can define a view *physics_fall_2017_watson* that lists the course ID and room number of all Physics courses offered in the Fall 2017 semester in the Watson building as follows:

```
create view physics_fall_2017_watson as
        select course_id, room_number
        from physics_fall_2017
        where building = 'Watson';
```

where *physics_fall_2017_watson* is itself a view relation. This is equivalent to:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = 2017)
    where building = 'Watson';
```

Telegram: @uni_k

www.konkur.in

4.2.3 Materialized Views

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

For example, consider the view *departments_total_salary*. If that view is materialized, its results would be stored in the database, allowing queries that use the view to potentially run much faster by using the precomputed view result, instead of recomputing it.

However, if an *instructor* tuple is added to or deleted from the *instructor* relation, the result of the query defining the view would change, and as a result the materialized view's contents must be updated. Similarly, if an instructor's salary is updated, the tuple in *departments_total_salary* corresponding to that instructor's department must be updated.

The process of keeping the materialized view up-to-date is called **materialized view maintenance** (or often, just **view maintenance**) and is covered in Section 16.5. View maintenance can be done immediately when any of the relations on which the view is defined is updated. Some database systems, however, perform view maintenance lazily, when the view is accessed. Some systems update materialized views only periodically; in this case, the contents of the materialized view may be stale, that is, not up-to-date, when it is used, and it should not be used if the application needs up-to-date data. And some database systems permit the database administrator to control which of the preceding methods is used for each materialized view.

Applications that use a view frequently may benefit if the view is materialized. Applications that demand fast response to certain queries that compute aggregates over large relations can also benefit greatly by creating materialized views corresponding to the queries. In this case, the aggregated result is likely to be much smaller than the large relations on which the view is defined; as a result the materialized view can be used to answer the query very quickly, avoiding reading the large underlying relations. The benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

SQL does not define a standard way of specifying that a view is materialized, but many database systems provide their own SQL extensions for this task. Some database systems always keep materialized views up-to-date when the underlying relations change, while others permit them to become out of date and periodically recompute them.

4.2.4 Update of a View

Although views are a useful tool for queries, they present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

Telegram: @uni_k

www.konkur.in

Suppose the view *faculty*, which we saw earlier, is made available to a clerk. Since we allow a view name to appear wherever a relation name is allowed, the clerk can write:

```
insert into faculty
    values ('30765', 'Green', 'Music');
```

This insertion must be represented by an insertion into the relation *instructor*, since *instructor* is the actual relation from which the database system constructs the view *faculty*. However, to insert a tuple into *instructor*, we must have some value for *salary*. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion, and return an error message to the user.
- Insert a tuple ('30765', 'Green', 'Music', *null*) into the *instructor* relation.

Another problem with modification of the database through views occurs with a view such as:

```
create view instructor_info as
select ID, name, building
from instructor, department
where instructor.dept_name = department.dept_name;
```

This view lists the *ID*, *name*, and building-name of each instructor in the university. Consider the following insertion through this view:

```
insert into instructor_info
    values ('69987', 'White', 'Taylor');
```

Suppose there is no instructor with ID 69987, and no department in the Taylor building. Then the only possible method of inserting tuples into the *instructor* and *department* relations is to insert ('69987', 'White', *null*, *null*) into *instructor* and (*null*, 'Taylor', *null*) into *department*. Then we obtain the relations shown in Figure 4.8. However, this update does not have the desired effect, since the view relation *instructor_info* still does *not* include the tuple ('69987', 'White', 'Taylor'). Thus, there is no way to update the relations *instructor* and *department* by using nulls to get the desired update on *instructor_info*.

Because of problems such as these, modifications are generally not permitted on view relations, except in limited cases. Different database systems specify different conditions under which they permit updates on view relations; see the database system manuals for details.

In general, an SQL view is said to be **updatable** (i.e., inserts, updates, or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- The **from** clause has only one database relation.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

instructor

Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

department

Figure 4.8 Relations *instructor* and *department* after insertion of tuples.

- The **select** clause contains only attribute names of the relation and does not have any expressions, aggregates, or **distinct** specification.
- Any attribute not listed in the **select** clause can be set to *null*; that is, it does not have a **not null** constraint and is not part of a primary key.
- The query does not have a **group by** or **having** clause.

Under these constraints, the **update**, **insert**, and **delete** operations would be allowed on the following view:

Telegram: @uni_k

www.konkur.in


```
create view history_instructors as
select *
from instructor
where dept_name = 'History';
```

Even with the conditions on updatability, the following problem still remains. Suppose that a user tries to insert the tuple ('25566', 'Brown', 'Biology', 100000) into the *history_instructors* view. This tuple can be inserted into the *instructor* relation, but it would not appear in the *history_instructors* view since it does not satisfy the selection imposed by the view.

By default, SQL would allow the above update to proceed. However, views can be defined with a **with check option** clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's **where** clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the **where** clause conditions.

SQL:1999 has a more complex set of rules about when inserts, updates, and deletes can be executed on a view that allows updates through a larger class of views; however, the rules are too complex to be discussed here.

An alternative, and often preferable, approach to modifying the database through a view is to use the trigger mechanism discussed in Section 5.3. The **instead of** feature in declaring triggers allows one to replace the default insert, update, and delete operations on a view with actions designed especially for each particular case.

4.3 Transactions

A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the first statement of the transaction was executed.

The keyword **work** is optional in both the statements.

Transaction rollback is useful if some error condition is detected during execution of a transaction. Commit is similar, in a sense, to saving changes to a document that is being edited, while rollback is similar to quitting the edit session without saving

Telegram: @uni_k

www.konkur.in

changes. Once a transaction has executed **commit work**, its effects can no longer be undone by **rollback work**. The database system guarantees that in the event of some failure, such as an error in one of the SQL statements, a power outage, or a system crash, a transaction's effects will be rolled back if it has not yet executed **commit work**. In the case of power outage or other system crash, the rollback occurs when the system restarts.

For instance, consider a banking application where we need to transfer money from one bank account to another in the same bank. To do so, we need to update two account balances, subtracting the amount transferred from one, and adding it to the other. If the system crashes after subtracting the amount from the first account but before adding it to the second account, the bank balances will be inconsistent. A similar problem occurs if the second account is credited before subtracting the amount from the first account and the system crashes just after crediting the amount.

As another example, consider our running example of a university application. We assume that the attribute *tot_cred* of each tuple in the *student* relation is kept up-to-date by modifying it whenever the student successfully completes a course. To do so, whenever the *takes* relation is updated to record successful completion of a course by a student (by assigning an appropriate grade), the corresponding *student* tuple must also be updated. If the application performing these two updates crashes after one update is performed, but before the second one is performed, the data in the database will be inconsistent.

By either committing the actions of a transaction after all its steps are completed, or rolling back all its actions in case the transaction could not complete all its actions successfully, the database provides an abstraction of a transaction as being **atomic**, that is, indivisible. Either all the effects of the transaction are reflected in the database or none are (after rollback).

Applying the notion of transactions to the above applications, the update statements should be executed as a single transaction. An error while a transaction executes one of its statements would result in undoing the effects of the earlier statements of the transaction so that the database is not left in a partially updated state.

If a program terminates without executing either of these commands, the updates are either committed or rolled back. The standard does not specify which of the two happens, and the choice is implementation dependent.

In many SQL implementations, including MySQL and PostgreSQL, by default each SQL statement is taken to be a transaction on its own, and it gets committed as soon as it is executed. Such *automatic commit* of individual SQL statements must be turned off if a transaction consisting of multiple SQL statements needs to be executed. How to turn off automatic commit depends on the specific SQL implementation, although many databases support the command **set autocommit off**.[6]

[6]There is a standard way of turning autocommit on or off when using application program interfaces such as JDBC or ODBC, which we study in Section 5.1.1 and Section 5.1.3, respectively.

Telegram: @uni_k

www.konkur.in

A better alternative, which is part of the SQL:1999 standard is to allow multiple SQL statements to be enclosed between the keywords **begin atomic** … **end**. All the statements between the keywords then form a single transaction, which is committed by default if execution reaches the **end** statement. Only some databases, such as SQL Server, support the above syntax. However, several other databases, such as MySQL and PostgreSQL, support a **begin** statement which starts a transaction containing all subsequent SQL statements, but do not support the **end** statement; instead, the transaction must be ended by either a **commit work** or a **rollback work** command.

If you use a database such as Oracle, where the automatic commit is not the default for DML statements, be sure to issue a **commit** command after adding or modifying data, or else when you disconnect, all your database modifications will be rolled back![7] You should be aware that although Oracle has automatic commit turned off by default, that default may be overridden by local configuration settings.

We study further properties of transactions in Chapter 17; issues in implementing transactions are addressed in Chapter 18 and Chapter 19.

4.4 Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database. This is in contrast to *security constraints*, which guard against access to the database by unauthorized users.

Examples of integrity constraints are:

- An instructor name cannot be *null*.
- No two instructors can have the same instructor ID.
- Every department name in the *course* relation must have a matching department name in the *department* relation.
- The budget of a department must be greater than $0.00.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, most database systems allow one to specify only those integrity constraints that can be tested with minimal overhead.

We have already seen some forms of integrity constraints in Section 3.2.2. We study some more forms of integrity constraints in this section. In Chapter 7, we study another form of integrity constraint, called **functional dependencies**, that is used primarily in the process of schema design.

[7] Oracle does automatically commit DDL statements.

Telegram: @uni_k

www.konkur.in

Integrity constraints are usually identified as part of the database schema design process and declared as part of the **create table** command used to create relations. However, integrity constraints can also be added to an existing relation by using the command **alter table** *table-name* **add** *constraint*, where *constraint* can be any constraint on the relation. When such a command is executed, the system first ensures that the relation satisfies the specified constraint. If it does, the constraint is added to the relation; if not, the command is rejected.

4.4.1 Constraints on a Single Relation

We described in Section 3.2 how to define tables using the **create table** command. The **create table** command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the **create table** command. The allowed integrity constraints include

- **not null**
- **unique**
- **check**(<predicate>)

We cover each of these types of constraints in the following sections.

4.4.2 Not Null Constraint

As we discussed in Chapter 3, the null value is a member of all domains, and as a result it is a legal value for every attribute in SQL by default. For certain attributes, however, null values may be inappropriate. Consider a tuple in the *student* relation where *name* is *null*. Such a tuple gives student information for an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be *null*. In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes *name* and *budget* to exclude null values, by declaring it as follows:

name **varchar**(20) **not null**
budget **numeric**(12,2) **not null**

The **not null** constraint prohibits the insertion of a null value for the attribute, and is an example of a **domain constraint**. Any database modification that would cause a null to be inserted in an attribute declared to be **not null** generates an error diagnostic.

There are many situations where we want to avoid null values. In particular, SQL prohibits null values in the primary key of a relation schema. Thus, in our university example, in the *department* relation, if the attribute *dept_name* is declared as the primary key for *department*, it cannot take a null value. As a result it would not need to be declared explicitly to be **not null**.

Telegram: @uni_k

www.konkur.in

4.4.3 Unique Constraint

SQL also supports an integrity constraint:

$$\textbf{unique}\ (A_{j_1}, A_{j_2}, \ldots, A_{j_m})$$

The **unique** specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ form a superkey; that is, no two tuples in the relation can be equal on all the listed attributes. However, attributes declared as unique are permitted to be *null* unless they have explicitly been declared to be **not null**. Recall that a null value does not equal any other value. (The treatment of nulls here is the same as that of the **unique** construct defined in Section 3.8.4.)

4.4.4 The Check Clause

When applied to a relation declaration, the clause **check**(P) specifies a predicate P that must be satisfied by every tuple in a relation.

A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause **check** (*budget* > 0) in the **create table** command for relation *department* would ensure that the value of **budget** is nonnegative.

As another example, consider the following:

```
create table section
   (course_id      varchar (8),
    sec_id         varchar (8),
    semester       varchar (6),
    year           numeric (4,0),
    building       varchar (15),
    room_number    varchar (7),
    time_slot_id   varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

Here, we use the **check** clause to simulate an enumerated type by specifying that *semester* must be one of 'Fall', 'Winter', 'Spring', or 'Summer'. Thus, the **check** clause permits attribute domains to be restricted in powerful ways that most programming-language type systems do not permit.

Null values present an interesting special case in the evaluation of a **check** clause. A **check** clause is satisfied if it is not false, so clauses that evaluate to **unknown** are not violations. If null values are not desired, a separate **not null** constraint (see Section 4.4.2) must be specified.

A **check** clause may appear on its own, as shown above, or as part of the declaration of an attribute. In Figure 4.9, we show the **check** constraint for the *semester* attribute

Telegram: @uni_k

www.konkur.in


```
create table classroom
    (building       varchar (15),
     room_number    varchar (7),
     capacity       numeric (4,0),
     primary key (building, room_number));

create table department
    (dept_name      varchar (20),
     building       varchar (15),
     budget         numeric (12,2) check (budget > 0),
     primary key (dept_name));

create table course
    (course_id      varchar (8),
     title          varchar (50),
     dept_name      varchar (20),
     credits        numeric (2,0) check (credits > 0),
     primary key (course_id),
     foreign key (dept_name) references department);

create table instructor
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     salary         numeric (8,2) check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department);

create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6) check (semester in
                                ('Fall', 'Winter', 'Spring', 'Summer')),
     year           numeric (4,0) check (year > 1759 and year < 2100),
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course,
     foreign key (building, room_number) references classroom);
```

Figure 4.9 SQL data definition for part of the university database.

Telegram: @uni_k

www.konkur.in

as part of the declaration of *semester*. The placement of a **check** clause is a matter of coding style. Typically, constraints on the value of a single attribute are listed with that attribute, while more complex **check** clauses are listed separately at the end of a **create table** statement.

The predicate in the **check** clause can, according to the SQL standard, be an arbitrary predicate that can include a subquery. However, currently none of the widely used database products allows the predicate to contain a subquery.

4.4.5 Referential Integrity

Often, we wish to ensure that a value that appears in one relation (the *referencing* relation) for a given set of attributes also appears for a certain set of attributes in another relation (the *referenced* relation). As we saw earlier, in Section 2.3, such conditions are called *referential integrity constraints*, and *foreign keys* are a form of a referential integrity constraint where the referenced attributes form a primary key of the referenced relation.

Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause, as we saw in Section 3.2.2. We illustrate foreign-key declarations by using the SQL DDL definition of part of our university database, shown in Figure 4.9. The definition of the *course* table has a declaration

"**foreign key** (*dept_name*) **references** *department*".

This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the *department* relation. Without this constraint, it is possible for a course to specify a nonexistent department name.

By default, in SQL a foreign key references the primary-key attributes of the referenced table. SQL also supports a version of the **references** clause where a list of attributes of the referenced relation can be specified explicitly.[8] For example, the foreign key declaration for the *course* relation can be specified as:

foreign key (*dept_name*) **references** *department*(*dept_name*)

The specified list of attributes must, however, be declared as a superkey of the referenced relation, using either a **primary key** constraint or a **unique** constraint. A more general form of a referential-integrity constraint, where the referenced columns need not be a candidate key, cannot be directly specified in SQL. The SQL standard specifies other constructs that can be used to implement such constraints, which are described in Section 4.4.8; however, these alternative constructs are not supported by any of the widely used database systems.

Note that the foreign key must reference a compatible set of attributes, that is, the number of attributes must be the same and the data types of corresponding attributes must be compatible.

[8] Some systems, notably MySQL, do not support the default and require that the attributes of the referenced relations be specified.

Telegram: @uni_k

www.konkur.in

We can use the following as part of a table definition to declare that an attribute forms a foreign key:

dept_name **varchar**(20) **references** *department*

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (i.e., the transaction performing the update action is rolled back). However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation *course*:

```
create table course
    ( …
    foreign key (dept_name) references department
                    on delete cascade
                    on update cascade,
    … );
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "**cascades**" to the *course* relation, deleting the tuple that refers to the department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *dept_name* in the referencing tuples in *course* to the new value as well. SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept_name*) can be set to *null* (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

If there is a chain of foreign-key dependencies across multiple relations, a deletion or update at one end of the chain can propagate across the entire chain. An interesting case where the **foreign key** constraint on a relation references the same relation appears in Exercise 4.9. If a cascading update or delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.

Null values complicate the semantics of referential-integrity constraints in SQL. Attributes of foreign keys are allowed to be *null*, provided that they have not otherwise been declared to be **not null**. If all the columns of a foreign key are nonnull in a given tuple, the usual definition of foreign-key constraints is used for that tuple. If any of the foreign-key columns is *null*, the tuple is defined automatically to satisfy the constraint. This definition may not always be the right choice, so SQL also provides constructs that allow you to change the behavior with null values; we do not discuss the constructs here.

Telegram: @uni_k

www.konkur.in

4.4.6 Assigning Names to Constraints

It is possible for us to assign a name to integrity constraints. Such names are useful if we want to drop a constraint that was defined previously.

To name a constraint, we precede the constraint with the keyword **constraint** and the name we wish to assign it. So, for example, if we wish to assign the name *minsalary* to the **check** constraint on the *salary* attribute of *instructor* (see Figure 4.9), we would modify the declaration for *salary* to:

salary **numeric**(8,2), **constraint** *minsalary* **check** (*salary* > 29000),

Later, if we decide we no longer want this constraint, we can write:

alter table *instructor* **drop constraint** *minsalary*;

Lacking a name, we would need first to use system-specific features to identify the system-assigned name for the constraint. Not all systems support this, but, for example, in Oracle, the system table *user_constraints* contains this information.

4.4.7 Integrity Constraint Violation During a Transaction

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation. For instance, suppose we have a relation *person* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *person*. That is, the constraint says that the *spouse* attribute must contain a name that is present in the *person* table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, in the preceding relation, with the spouse attributes set to Mary and John, respectively. The insertion of the first tuple would violate the foreign-key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted, the foreign-key constraint would hold again.

To handle such situations, the SQL standard allows a clause **initially deferred** to be added to a constraint specification; the constraint would then be checked at the end of a transaction and not at intermediate steps. A constraint can alternatively be specified as **deferrable**, which means it is checked immediately by default but can be deferred when desired. For constraints declared as deferrable, executing a statement **set constraints** *constraint-list* **deferred** as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction. Constraints that are to appear in a constraint list must have names assigned. The default behavior is to check constraints immediately, and many database implementations do not support deferred constraint checking.

We can work around the problem in the preceding example in another way, if the *spouse* attribute can be set to *null*: We set the spouse attributes to *null* when inserting the

Telegram: @uni_k

www.konkur.in

tuples for John and Mary, and we update them later. However, this technique requires more programming effort, and it does not work if the attributes cannot be set to *null*.

4.4.8 Complex Check Conditions and Assertions

There are additional constructs in the SQL standard for specifying integrity constraints that are not currently supported by most systems. We discuss some of these in this section.

As defined by the SQL standard, the predicate in the **check** clause can be an arbitrary predicate that can include a subquery. If a database implementation supports subqueries in the **check** clause, we could specify the following referential-integrity constraint on the relation *section*:

check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

The **check** condition verifies that the *time_slot_id* in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time_slot* changes (in this case, when a tuple is deleted or modified in relation *time_slot*).

Another natural constraint on our university schema would be to require that every section has at least one instructor teaching the section. In an attempt to enforce this, we may try to declare that the attributes (*course_id*, *sec_id*, *semester*, *year*) of the *section* relation form a foreign key referencing the corresponding attributes of the *teaches* relation. Unfortunately, these attributes do not form a candidate key of the relation *teaches*. A check constraint similar to that for the *time_slot* attribute can be used to enforce this constraint, if check constraints with subqueries were supported by a database system.

Complex **check** conditions can be useful when we want to ensure the integrity of data, but they may be costly to test. In our example, the predicate in the **check** clause would not only have to be evaluated when a modification is made to the *section* relation, but it may have to be checked if a modification is made to the *time_slot* relation because that relation is referenced in the subquery.

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Consider the following constraints, which can be expressed using assertions.

- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot.[9]

[9] We assume that lectures are not displayed remotely in a second classroom! An alternative constraint that specifies that "an instructor cannot teach two courses in a given semester in the same time slot" may not hold since courses are sometimes cross-listed; that is, the same course is given two identifiers and titles.

Telegram: @uni_k

www.konkur.in


```
create assertion credits_earned_constraint check
(not exists (select ID
             from student
             where tot_cred <> (select coalesce(sum(credits), 0)
                                from takes natural join course
                                where student.ID= takes.ID
                                      and grade is not null and grade<> 'F' )))
```

Figure 4.10 An assertion example.

An assertion in SQL takes the form:

create assertion <assertion-name> **check** <predicate>;

In Figure 4.10, we show how the first example of constraints can be written in SQL. Since SQL does not provide a "for all X, $P(X)$" construct (where P is a predicate), we are forced to implement the constraint by an equivalent construct, "not exists X such that not $P(X)$", that can be expressed in SQL.

We leave the specification of the second constraint as an exercise. Although these two constraints can be expressed using **check** predicates, using an assertion may be more natural, especially for the second constraint.

When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated. This testing may introduce a significant amount of overhead if complex assertions have been made. Hence, assertions should be used with great care. The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions, or to provide specialized forms of assertion that are easier to test.

Currently, none of the widely used database systems supports either subqueries in the **check** clause predicate or the **create assertion** construct. However, equivalent functionality can be implemented using triggers, which are described in Section 5.3, if they are supported by the database system. Section 5.3 also describes how the referential integrity constraint on *time_slot_id* can be implemented using triggers.

4.5 SQL Data Types and Schemas

In Chapter 3, we covered a number of built-in data types supported in SQL, such as integer types, real types, and character types. There are additional built-in data types supported by SQL, which we describe below. We also describe how to create basic user-defined types in SQL.

Telegram: @uni_k

www.konkur.in

4.5.1 Date and Time Types in SQL

In addition to the basic data types we introduced in Section 3.2, the SQL standard supports several data types relating to dates and times:

- **date**: A calendar date containing a (four-digit) year, month, and day of the month.
- **time**: The time of day, in hours, minutes, and seconds. A variant, **time**(p), can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time-zone information along with the time by specifying **time with timezone**.
- **timestamp**: A combination of **date** and **time**. A variant, **timestamp**(p), can be used to specify the number of fractional digits for seconds (the default here being 6). Time-zone information is also stored if **with timezone** is specified.

Date and time values can be specified like this:

date '2018-04-25'
time '09:30:00'
timestamp '2018-04-25 10:29:01.45'

Dates must be specified in the format year followed by month followed by day, as shown.[10] The seconds field of **time** or **timestamp** can have a fractional part, as in the timestamp above.

To extract individual fields of a **date** or **time** value d, we can use **extract** (*field* **from** d), where *field* can be one of **year, month, day, hour, minute**, or **second**. Time-zone information can be extracted using **timezone_hour** and **timezone_minute**.

SQL defines several functions to get the current date and time. For example, **current_date** returns the current date, **current_time** returns the current time (with time zone), and **localtime** returns the current local time (without time zone). Timestamps (date plus time) are returned by **current_timestamp** (with time zone) and **localtimestamp** (local date and time without time zone).

Some systems, including MySQL offer the **datetime** data type that represents a time that is not adjustable for time zone. In practice, specification of time has numerous special cases, including the use of standard time versus "daylight" or "summer" time. Systems vary in the range of times representable.

SQL allows comparison operations on all the types listed here, and it allows both arithmetic and comparison operations on the various numeric types. SQL also provides a data type called **interval**, and it allows computations based on dates and times and on intervals. For example, if x and y are of type **date**, then $x - y$ is an interval whose value is the number of days from date x to date y. Similarly, adding or subtracting an interval from a date or time gives back a date or time, respectively.

[10] Many database systems offer greater flexibility in default conversions of

Telegram: @uni_k

www.konkur.in

4.5.2 Type Conversion and Formatting Functions

Although systems perform some data type **conversions** automatically, others need to be requested explicitly. We can use an expression of the form **cast** (*e* **as** *t*) to convert an expression *e* to the type *t*. Data-type conversions may be needed to perform certain operations or to enforce certain sort orders. For example, consider the *ID* attribute of *instructor*, which we have specified as being a string (**varchar**(5)). If we were to order output by this attribute, the ID 11111 comes before the ID 9, because the first character, '1', comes before '9'. However, if we were to write:

select cast(*ID* **as numeric**(5)) **as** *inst_id*
from *instructor*
order by *inst_id*

the result would be the sorted order we desire.

A different type of conversion may be required for data to be displayed as the result of a query. For example, we may wish numbers to be shown with a specific number of digits, or data to be displayed in a particular format (such as month-day-year or day-month-year). These changes in display format are not conversion of data type but rather conversion of format. Database systems offer a variety of formatting functions, and details vary among the leading systems. MySQL offers a **format** function. Oracle and PostgreSQL offer a set of functions, **to_char, to_number,** and **to_date**. SQL Server offers a **convert** function.

Another issue in displaying results is the handling of null values. In this text, we use *null* for clarity of reading, but the default in most systems is just to leave the field blank. We can choose how null values are output in a query result using the **coalesce** function. It takes an arbitrary number of arguments, all of which must be of the same type, and returns the first non-null argument. For example, if we wished to display instructor IDs and salaries but to show null salaries as 0, we would write:

select *ID*, **coalesce**(*salary*, 0) **as** *salary*
from *instructor*

A limitation of **coalesce** is the requirement that all the arguments must be of the same type. If we had wanted null salaries to appear as 'N/A' to indicate "not available", we would not be able to use **coalesce**. System-specific functions, such as Oracle's **decode**, do allow such conversions. The general form of **decode** is:

decode (*value, match-1, replacement-1, match-2, replacement-2, …, match-N, replacement-N, default-replacement*);

It compares *value* against the *match* values and if a match is found, it replaces the attribute value with the corresponding replacement value. If no match succeeds, then the attribute value is replaced with the default replacement value. There are no require-

Telegram: @uni_k

www.konkur.in

ments that datatypes match. Conveniently, the value *null* may appear as a *match* value and, unlike the usual case, *null* is treated as being equal to *null*. Thus, we could replace null salaries with 'N/A' as follows:

```
select ID, decode (salary, null, 'N/A', salary) as salary
from instructor
```

4.5.3 Default Values

SQL allows a **default** value to be specified for an attribute as illustrated by the following **create table** statement:

```
create table student
    (ID            varchar (5),
     name          varchar (20) not null,
     dept_name     varchar (20),
     tot_cred      numeric (3,0) default 0,
     primary key (ID));
```

The default value of the *tot_cred* attribute is declared to be 0. As a result, when a tuple is inserted into the *student* relation, if no value is provided for the *tot_cred* attribute, its value is set to 0. The following **insert** statement illustrates how an insertion can omit the value for the *tot_cred* attribute.

```
insert into student(ID, name, dept_name)
    values ('12789', 'Newman', 'Comp. Sci.');
```

4.5.4 Large-Object Types

Many database applications need to store attributes whose domain consists of large data items such as a photo, a high-resolution medical image, or a video. SQL, therefore, provides **large-object data types** for character data (**clob**) and binary data (**blob**). The letters "lob" in these data types stand for "Large OBject." For example, we may declare attributes

```
book_review clob(10KB)
image blob(10MB)
movie blob(2GB)
```

For result tuples containing large objects (multiple megabytes to gigabytes), it is inefficient or impractical to retrieve an entire large object into memory. Instead, an application would usually use an SQL query to retrieve a "locator" for a large object and then use the locator to manipulate the object from the host language in which the application itself is written. For instance, the JDBC application program interface (described in Section 5.1.1) permits a locator to be fetched instead of the entire large

Telegram: @uni_k

www.konkur.in



Telegram: @uni_k

www.konkur.in

object; the locator can then be used to fetch the large object in small pieces, rather than all at once, much like reading data from an operating system file using a `read` function call.

4.5.5 User-Defined Types

SQL supports two forms of **user-defined data types**. The first form, which we cover here, is called **distinct types**. The other form, called **structured data types**, allows the creation of complex data types with nested record structures, arrays, and multisets. We do not cover structured data types in this chapter, but we describe them in Section 8.2.

It is possible for several attributes to have the same data type. For example, the *name* attributes for student name and instructor name might have the same domain: the set of all person names. However, the domains of *budget* and *dept_name* certainly ought to be distinct. It is perhaps less clear whether *name* and *dept_name* should have the same domain. At the implementation level, both instructor names and department names are character strings. However, we would normally not consider the query "Find all instructors who have the same name as a department" to be a meaningful query. Thus, if we view the database at the conceptual, rather than the physical, level, *name* and *dept_name* should have distinct domains.

More importantly, at a practical level, assigning an instructor's name to a department name is probably a programming error; similarly, comparing a monetary value expressed in dollars directly with a monetary value expressed in pounds is also almost surely a programming error. A good type system should be able to detect such assignments or comparisons. To support such checks, SQL provides the notion of **distinct types**.

The **create type** clause can be used to define new types. For example, the statements:

create type *Dollars* **as numeric**(12,2) **final**;
create type *Pounds* **as numeric**(12,2) **final**;

define the user-defined types *Dollars* and *Pounds* to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point.[11] The newly created types can then be used, for example, as types of attributes of relations. For example, we can declare the *department* table as:

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget *Dollars*);

An attempt to assign a value of type *Dollars* to a variable of type *Pounds* results in a compile-time error, although both are of the same numeric type. Such an assignment is likely to be due to a programmer error, where the programmer forgot about the

[11] The keyword **final** isn't really meaningful in this context but is required by the SQL:1999 standard for reasons we won't get into here; some implementations allow the **final** keyword to be omitted.

Telegram: @uni_k

www.konkur.in

differences in currency. Declaring different types for different currencies helps catch such errors.

As a result of strong type checking, the expression (*department.budget*+20) would not be accepted since the attribute and the integer constant 20 have different types. As we saw in Section 4.5.2, values of one type can be converted to another domain, as illustrated below:

cast (*department.budget* **to** *numeric*(12,2))

We could do addition on the numeric type, but to save the result back to an attribute of type *Dollars* we would have to use another cast expression to convert the type back to *Dollars*.

SQL provides **drop type** and **alter type** clauses to drop or modify types that have been created earlier.

Even before user-defined types were added to SQL (in SQL:1999), SQL had a similar but subtly different notion of **domain** (introduced in SQL-92), which can add integrity constraints to an underlying type. For example, we could define a domain *DDollars* as follows.

create domain *DDollars* **as numeric**(12,2) **not null**;

The domain *DDollars* can be used as an attribute type, just as we used the type *Dollars*. However, there are two significant differences between types and domains:

1. Domains can have constraints, such as **not null**, specified on them, and can have default values defined for variables of the domain type, whereas user-defined types cannot have constraints or default values specified on them. User-defined types are designed to be used not just for specifying attribute types, but also in procedural extensions to SQL where it may not be possible to enforce constraints.
2. Domains are not strongly typed. As a result, values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.

When applied to a domain, the **check** clause permits the schema designer to specify a predicate that must be satisfied by any attribute declared to be from this domain. For instance, a **check** clause can ensure that an instructor's salary domain allows only values greater than a specified value:

create domain *YearlySalary* **numeric**(8,2)
constraint *salary_value_test* **check**(**value** >= 29000.00);

The domain *YearlySalary* has a constraint that ensures that the *YearlySalary* is greater than or equal to $29,000.00. The clause **constraint** *salary value test* is optional and is

Telegram: @uni_k

www.konkur.in



used to give the name *salary_value_test* to the constraint. The name is used by the system to indicate the constraint that an update violated.

As another example, a domain can be restricted to contain only a specified set of values by using the **in** clause:

```
create domain degree_level varchar(10)
        constraint degree_level_test
                check (value in ('Bachelors', 'Masters', 'Doctorate'));
```

4.5.6 Generating Unique Key Values

In our university example, we have seen primary-key attributes with different data types. Some, like *dept_name*, hold actual real-world information. Others, like *ID*, hold values created by the enterprise solely for identification purposes. Those latter types of primary-key domains generate the practical problem of new-value creation. Suppose

Telegram: @uni_k

www.konkur.in

the university hires a new instructor. What ID should be assigned? How do we determine that the new ID is unique? Although it is possible to write an SQL statement to do this, such a statement would need to check all preexisting IDs, which would harm system performance. Alternatively, one could set up a special table holding the largest ID value issued so far. Then, when a new ID is needed, that value can be incremented to the next one in sequence and stored as the new largest value.

Database systems offer automatic management of unique key-value generation. The syntax differs among the most popular systems and, sometimes, between versions of systems. The syntax we show here is close to that of Oracle and DB2. Suppose that instead of declaring instructor IDs in the *instructor* relation as "*ID* **varchar**(5)", we instead choose to let the system select a unique instructor ID value. Since this feature works only for numeric key- value data types, we change the type of *ID* to **number**, and write:

ID **number**(5) **generated always as identity**

When the **always** option is used, any **insert** statement must avoid specifying a value for the automatically generated key. To do this, use the syntax for **insert** in which the attribute order is specified (see Section 3.9.2). For our example of *instructor*, we need specify only the values for *name*, *dept_name*, and *salary*, as shown in the following example:

insert into *instructor* (*name*, *dept_name*, *salary*)
values ('Newprof', 'Comp. Sci.', 100000);

The generated ID value can be found via a normal **select** query. If we replace **always** with **by default**, we have the option of specifying our own choice of *ID* or relying on the system to generate one.

In PostgreSQL, we can define the type of *ID* as **serial**, which tells PostgreSQL to automatically generate identifiers; in MySQL we use **auto_increment** in place of **generated always as identity**, while in SQL Server we can use just **identity**.

Additional options can be specified, with the **identity** specification, depending on the database, including setting minimum and maximum values, choosing the starting value, choosing the increment from one value to the next, and so on.

Further, many databases support a **create sequence** construct, which creates a sequence counter object separate from any relation, and allow SQL queries to get the next value from the sequence. Each call to get the next value increments the sequence counter. See the system manuals of the database to find the exact syntax for creating sequences, and for retrieving the next value. Using sequences, we can generate identifiers that are unique across multiple relations, for example, across *student.ID*, and *instructor.ID*.

Telegram: @uni_k

www.konkur.in

4.5.7 Create Table Extensions

Applications often require the creation of tables that have the same schema as an existing table. SQL provides a **create table like** extension to support this task:[12]

create table *temp_instructor* **like** *instructor*;

The above statement creates a new table *temp_instructor* that has the same schema as *instructor*.

When writing a complex query, it is often useful to store the result of a query as a new table; the table is usually temporary. Two statements are required, one to create the table (with appropriate columns) and the second to insert the query result into the table. SQL:2003 provides a simpler technique to create a table containing the results of a query. For example, the following statement creates a table *t1* containing the results of a query.

```
create table t1 as
        (select *
        from instructor
        where dept_name = 'Music')
with data;
```

By default, the names and data types of the columns are inferred from the query result. Names can be explicitly given to the columns by listing the column names after the relation name.

As defined by the SQL:2003 standard, if the **with data** clause is omitted, the table is created but not populated with data. However, many implementations populate the table with data by default even if the **with data** clause is omitted. Note that several implementations support the functionality of **create table** … **like** and **create table** … **as** using different syntax; see the respective system manuals for further details.

The above **create table** … **as** statement, closely resembles the **create view** statement and both are defined by using queries. The main difference is that the contents of the table are set when the table is created, whereas the contents of a view always reflect the current query result.

4.5.8 Schemas, Catalogs, and Environments

To understand the motivation for schemas and catalogs, consider how files are named in a file system. Early file systems were flat; that is, all files were stored in a single directory. Current file systems have a directory (or, synonymously, folder) structure, with files stored within subdirectories. To name a file uniquely, we must specify the full path name of the file, for example, /users/avi/db-book/chapter3.tex.

[12]This syntax is not supported in all systems.

Telegram: @uni_k

www.konkur.in

Like early file systems, early database systems also had a single name space for all relations. Users had to coordinate to make sure they did not try to use the same name for different relations. Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**. (Some database implementations use the term *d*atabase in place of the term *c*atalog.)

In order to perform any actions on a database, a user (or a program) must first *connect* to the database. The user must provide the user name and usually, a password for verifying the identity of the user. Each user has a default catalog and schema, and the combination is unique to the user. When a user connects to a database system, the default catalog and schema are set up for the connection; this corresponds to the current directory being set to the user's home directory when the user logs into an operating system.

To identify a relation uniquely, a three-part name may be used, for example,

catalog5.univ_schema.course

We may omit the catalog component, in which case the catalog part of the name is considered to be the default catalog for the connection. Thus, if *catalog5* is the default catalog, we can use *univ_schema.course* to identify the same relation uniquely.

If a user wishes to access a relation that exists in a different schema than the default schema for that user, the name of the schema must be specified. However, if a relation is in the default schema for a particular user, then even the schema name may be omitted. Thus, we can use just *course* if the default catalog is *catalog5* and the default schema is *univ_schema*.

With multiple catalogs and schemas available, different applications and different users can work independently without worrying about name clashes. Moreover, multiple versions of an application—one a production version, other test versions—can run on the same database system.

The default catalog and schema are part of an **SQL environment** that is set up for each connection. The environment additionally contains the user identifier (also referred to as the *authorization identifier*). All the usual SQL statements, including the DDL and DML statements, operate in the context of a schema.

We can create and drop schemas by means of **create schema** and **drop schema** statements. In most database systems, schemas are also created automatically when user accounts are created, with the schema name set to the user account name. The schema is created in either a default catalog or a catalog specified when creating the user account. The newly created schema becomes the default schema for the user account.

Creation and dropping of catalogs is implementation dependent and not part of the SQL standard.

Telegram: @uni_k

www.konkur.in

4.6 Index Definition in SQL

Many queries reference only a small proportion of the records in a file. For example, a query like "Find all instructors in the Physics department" or "Find the *salary* value of the instructor with *ID* 22201" references only a fraction of the instructor records. It is inefficient for the system to read every record and to check *ID* field for the *ID* "32556," or the *building* field for the value "Physics".

An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation. For example, if we create an index on attribute *dept_name* of relation *instructor*, the database system can find the record with any specified *dept_name* value, such as "Physics", or "Music", directly, without reading all the tuples of the *instructor* relation. An index can also be created on a list of attributes, for example, on attributes *name* and *dept_name* of *instructor*.

Indices are not required for correctness, since they are redundant data structures. Indices form part of the physical schema of the database, as opposed to its logical schema.

However, indices are important for efficient processing of transactions, including both update transactions and queries. Indices are also important for efficient enforcement of integrity constraints such as primary-key and foreign-key constraints. In principle, a database system can decide automatically what indices to create. However, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain.

Therefore, most SQL implementations provide the programmer with control over the creation and removal of indices via data-definition-language commands. We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL standard. The SQL standard does not support control of the physical database schema; it restricts itself to the logical database schema.

We create an index with the **create index** command, which takes the form:

create index <index-name> **on** <relation-name> (<attribute-list>);

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index named *dept_index* on the *instructor* relation with *dept_name* as the search key, we write:

create index *dept_index* **on** *instructor* (*dept_name*);

When a user submits an SQL query that can benefit from using an index, the SQL query processor automatically uses the index. For example, given an query that

Telegram: @uni_k

www.konkur.in

selects the *instructor* tuple with *dept_name* "Music", the SQL query processor would use the index *dept_index* defined above to find the required tuple without reading the whole relation.

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command:

create unique index *dept_index* **on** *instructor* (*dept_name*);

declares *dept_name* to be a candidate key for *instructor* (which is probably not what we actually would want for our university database). If, at the time we enter the **create unique index** command, *dept_name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

drop index <index-name>;

Many database systems also provide a way to specify the type of index to be used, such as B^+-tree or hash indices, which we study in Chapter 14. Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search key of the clustered index. We study in Chapter 14 how indices are actually implemented, as well as what indices are automatically created by databases, and how to decide on what additional indices to create.

4.7 Authorization

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Telegram: @uni_k

www.konkur.in

When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. If the query or update is not authorized, it is rejected.

In addition to authorizations on data, users may also be granted authorizations on the database schema, allowing them, for example, to create, modify, or drop relations. A user who has some form of authorization may be allowed to pass on (grant) this authorization to other users, or to withdraw (revoke) an authorization that was granted earlier. In this section, we see how each of these authorizations can be specified in SQL.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, and so on. This form of authorization is analogous to that of a **superuser**, administrator, or operator for an operating system.

4.7.1 Granting and Revoking of Privileges

The SQL standard includes the **privileges** **select**, **insert**, **update**, and **delete**. The privilege **all privileges** can be used as a short form for all the allowable privileges. A user who creates a new relation is given all privileges on that relation automatically.

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

grant <privilege list>
on <relation name or view name>
to <user/role list>;

The *privilege list* allows the granting of several privileges in one command. The notion of roles is covered in Section 4.7.2.

The **select** authorization on a relation is required to read tuples in the relation. The following **grant** statement grants database users Amit and Satoshi **select** authorization on the *department* relation:

grant select on *department* **to** *Amit*, *Satoshi*;

This allows those users to run queries on the *department* relation.

The **update** authorization on a relation allows a user to update any tuple in the relation. The **update** authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.

This **grant** statement gives users Amit and Satoshi update authorization on the *budget* attribute of the *department* relation:

Telegram: @uni_k

www.konkur.in

grant update (*budget*) **on** *department* **to** *Amit*, *Satoshi*;

The **insert** authorization on a relation allows a user to insert tuples into the relation. The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to *null*.

The **delete** authorization on a relation allows a user to delete tuples from a relation.

The user name **public** refers to all current and future users of the system. Thus, privileges granted to **public** are implicitly granted to all current and future users.

By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. SQL allows a privilege grant to specify that the recipient may further grant the privilege to another user. We describe this feature in more detail in Section 4.7.5.

It is worth noting that the SQL authorization mechanism grants privileges on an entire relation, or on specified attributes of a relation. However, it does not permit authorizations on specific tuples of a relation.

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

revoke <privilege list>
on <relation name or view name>
from <user/role list>;

Thus, to revoke the privileges that we granted previously, we write

revoke select on *department* **from** *Amit*, *Satoshi*;
revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*;

Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user. We return to this issue in Section 4.7.5.

4.7.2 Roles

Consider the real-world roles of various people in a university. Each instructor must have the same types of authorizations on the same set of relations. Whenever a new instructor is appointed, she will have to be given all these authorizations individually.

A better approach would be to specify the authorizations that every instructor is to be given, and to identify separately which database users are instructors. The system can use these two pieces of information to determine the authorizations of each instructor. When a new instructor is hired, a user identifier must be allocated to him, and he must be identified as an instructor. Individual permissions given to instructors need not be specified again.

Telegram: @uni_k

www.konkur.in

The notion of **roles** captures this concept. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that she is authorized to perform.

In our university database, examples of roles could include *instructor*, *teaching_assistant*, *student*, *dean*, and *department_chair*.

A less preferable alternative would be to create an *instructor* userid and permit each instructor to connect to the database using the *instructor* userid. The problem with this approach is that it would not be possible to identify exactly which instructor carried out a database update, and this could create security risks. Furthermore, if an instructor leaves the university or is moved to a non instructional role, then a new *instructor* password must be created and distributed in a secure manner to all instructors. The use of roles has the benefit of requiring users to connect to the database with their own userid.

Any authorization that can be granted to a user can be granted to a role. Roles are granted to users just as authorizations are.

Roles can be created in SQL as follows:

create role *instructor*;

Roles can then be granted privileges just as the users can, as illustrated in this statement:

grant select on *takes*
to *instructor*;

Roles can be granted to users, as well as to other roles, as these statements show:

create role *dean*;
grant *instructor* **to** *dean*;
grant *dean* **to** Satoshi;

Thus, the privileges of a user or a role consist of:

- All privileges directly granted to the user/role.
- All privileges granted to roles that have been granted to the user/role.

Note that there can be a chain of roles; for example, the role *teaching_assistant* may be granted to all *instructors*. In turn, the role *instructor* is granted to all *dean*s. Thus, the *dean* role inherits all privileges granted to the roles *instructor* and to *teaching_assistant* in addition to privileges granted directly to *dean*.

When a user logs in to the database system, the actions executed by the user during that session have all the privileges granted directly to the user, as well as all privileges

Telegram: @uni_k

www.konkur.in

granted to roles that are granted (directly or indirectly via other roles) to that user. Thus, if a user Amit has been granted the role *dean*, user Amit holds all privileges granted directly to Amit, as well as privileges granted to *dean*, plus privileges granted to *instructor* and *teaching_assistant* if, as above, those roles were granted (directly or indirectly) to the role *dean*.

It is worth noting that the concept of role-based authorization is not specific to SQL, and role-based authorization is used for access control in a wide variety of shared applications.

4.7.3 Authorization on Views

In our university example, consider a staff member who needs to know the salaries of all faculty in a particular department, say the Geology department. This staff member is not authorized to see information regarding faculty in other departments. Thus, the staff member must be denied direct access to the *instructor* relation. But if he is to have access to the information for the Geology department, he might be granted access to a view that we shall call *geo_instructor*, consisting of only those *instructor* tuples pertaining to the Geology department. This view can be defined in SQL as follows:

```
create view geo_instructor as
    (select *
     from instructor
     where dept_name = 'Geology');
```

Suppose that the staff member issues the following SQL query:

```
select *
from geo_instructor;
```

The staff member is authorized to see the result of this query. However, when the query processor translates it into a query on the actual relations in the database, it replaces uses of a view by the definition of the view, producing a query on *instructor*. Thus, the system must check authorization on the clerk's query before it replaces views by their definitions.

A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user who creates a view cannot be given **update** authorization on a view without having **update** authorization on the relations used to define the view. If a user creates a view on which no authorization can be granted, the system will deny the view creation request. In our *geo_instructor* view example, the creator of the view must have **select** authorization on the *instructor* relation.

As we will see in Section 5.2, SQL supports the creation of functions and procedures, which may, in turn, contain queries and updates. The **execute** privilege can be granted on a function or procedure, enabling a user to execute the function or proce-

Telegram: @uni_k

www.konkur.in

dure. By default, just like views, functions and procedures have all the privileges that the creator of the function or procedure had. In effect, the function or procedure runs as if it were invoked by the user who created the function.

Although this behavior is appropriate in many situations, it is not always appropriate. Starting with SQL:2003, if the function definition has an extra clause **sql security invoker**, then it is executed under the privileges of the user who invokes the function, rather than the privileges of the **definer** of the function. This allows the creation of libraries of functions that can run under the same authorization as the invoker.

4.7.4 Authorizations on Schema

The SQL standard specifies a primitive authorization mechanism for the database schema: Only the owner of the schema can carry out any modification to the schema, such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices.

However, SQL includes a **references** privilege that permits a user to declare foreign keys when creating relations. The SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege. The following **grant** statement allows user Mariano to create relations that reference the key *dept_name* of the *department* relation as a foreign key:

grant references (*dept_name*) **on** *department* **to** Mariano;

Initially, it may appear that there is no reason ever to prevent users from creating foreign keys referencing another relation. However, recall that foreign-key constraints restrict deletion and update operations on the referenced relation. Suppose Mariano creates a foreign key in a relation *r* referencing the *dept_name* attribute of the *department* relation and then inserts a tuple into *r* pertaining to the Geology department. It is no longer possible to delete the Geology department from the *department* relation without also modifying relation *r*. Thus, the definition of a foreign key by Mariano restricts future activity by other users; therefore, there is a need for the **references** privilege.

Continuing to use the example of the *department* relation, the references privilege on *department* is also required to create a **check** constraint on a relation *r* if the constraint has a subquery referencing *department*. This is reasonable for the same reason as the one we gave for foreign-key constraints; a check constraint that references a relation limits potential updates to that relation.

4.7.5 Transfer of Privileges

A user who has been granted some form of authorization may be allowed to pass on this authorization to other users. By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate **grant** command. For example, if we wish to allow

Telegram: @uni_k

www.konkur.in

Amit the **select** privilege on *department* and allow Amit to grant this privilege to others, we write:

grant select on *department* **to** Amit **with grant option**;

The creator of an object (relation/view/role) holds all privileges on the object, including the privilege to grant privileges to others.

Consider, as an example, the granting of update authorization on the *teaches* relation of the university database. Assume that, initially, the database administrator grants update authorization on *teaches* to users U_1, U_2, and U_3, who may, in turn, pass on this authorization to other users. The passing of a specific authorization from one user to another can be represented by an **authorization graph**. The nodes of this graph are the users.

Consider the graph for update authorization on *teaches*. The graph includes an edge $U_i \rightarrow U_j$ if user U_i grants update authorization on *teaches* to U_j. The root of the graph is the database administrator. In the sample graph in Figure 4.11, observe that user U_5 is granted authorization by both U_1 and U_2; U_4 is granted authorization by only U_1.

A user has an authorization *if and only if* there is a path from the root of the authorization graph (the node representing the database administrator) down to the node representing the user.

4.7.6 Revoking of Privileges

Suppose that the database administrator decides to revoke the authorization of user U_1. Since U_4 has authorization from U_1, that authorization should be revoked as well. However, U_5 was granted authorization by both U_1 and U_2. Since the database administrator did not revoke update authorization on *teaches* from U_2, U_5 retains update

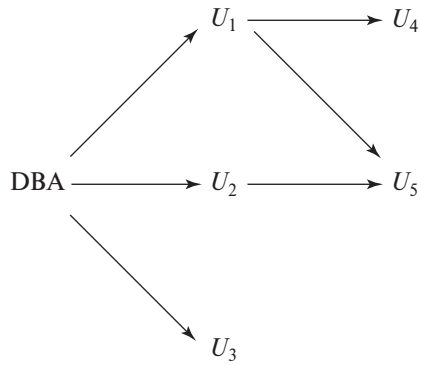


Figure 4.11 Authorization-grant graph (U_1, U_2, ..., U_5 are users and DBA refers to the database

Telegram: @uni_k

www.konkur.in

authorization on *teaches*. If U_2 eventually revokes authorization from U_5, then U_5 loses the authorization.

A pair of devious users might attempt to defeat the rules for revocation of authorization by granting authorization to each other. For example, U_2 is initially granted an authorization by the database administrator, and U_2 further grants it to U_3. Suppose U_3 now grants the privilege back to U_2. If the database administrator revokes authorization from U_2, it might appear that U_2 retains authorization through U_3. However, note that once the administrator revokes authorization from U_2, there is no path in the authorization graph from the root either to U_2 or to U_3. Thus, SQL ensures that the authorization is revoked from both the users.

As we just saw, revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called *cascading revocation*. In most database systems, cascading is the default behavior. However, the **revoke** statement may specify **restrict** in order to prevent cascading revocation:

revoke select on *department* **from** Amit, Satoshi **restrict**;

In this case, the system returns an error if there are any cascading revocations and does not carry out the revoke action.

The keyword **cascade** can be used instead of **restrict** to indicate that revocation should cascade; however, it can be omitted, as we have done in the preceding examples, since it is the default behavior.

The following **revoke** statement revokes only the grant option, rather than the actual **select** privilege:

revoke grant option for select on *department* **from** Amit;

Note that some database implementations do not support the above syntax; instead, the privilege itself can be revoked and then granted again without the grant option.

Cascading revocation is inappropriate in many situations. Suppose Satoshi has the role of *dean*, grants *instructor* to Amit, and later the role *dean* is revoked from Satoshi (perhaps because Satoshi leaves the university); Amit continues to be employed on the faculty and should retain the *instructor* role.

To deal with this situation, SQL permits a privilege to be granted by a role rather than by a user. SQL has a notion of the current role associated with a session. By default, the current role associated with a session is null (except in some special cases). The current role associated with a session can be set by executing **set role** *role_name*. The specified role must have been granted to the user, otherwise the **set role** statement fails.

To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

granted by current_role

to the grant statement, provided the current role is not null.

Telegram: @uni_k

www.konkur.in

Suppose the granting of the role *instructor* (or other privileges) to Amit is done using the **granted by current_role** clause, with the current role set to *dean*, instead of the grantor being the user Satoshi. Then, revoking of roles/privileges (including the role *dean*) from Satoshi will not result in revoking of privileges that had the grantor set to the role *dean*, even if Satoshi was the user who executed the grant; thus, Amit would retain the *instructor* role even after Satoshi's privileges are revoked.

4.7.7 Row-Level Authorization

The types of authorization privileges we have studied apply at the level of relations or views. Some database systems provide mechanisms for fine-grained authorization at the level of specific tuples within a relation.

Suppose, for example, that we wish to allow a student to see her or his own data in the *takes* relation but not those data of other users. We can enforce such a restriction using row-level authorization, if the database supports it. We describe row-level authorization in Oracle below; PostgreSQL and SQL Server too support row-level authorization using a conceptually similar mechanism, but using a different syntax.

The Oracle **Virtual Private Database** (**VPD**) feature supports row-level authorization as follows. It allows a system administrator to associate a function with a relation; the function returns a predicate that gets added automatically to any query that uses the relation. The predicate can use the function **sys_context**, which returns the identifier of the user on whose behalf a query is being executed. For our example of students accessing their data in the *takes* relation, we would specify the following predicate to be associated with the *takes* relation:

$$ID = \textbf{sys_context}\ (\text{'USERENV'}, \text{'SESSION_USER'})$$

This predicate is added by the system to the **where** clause of every query that uses the *takes* relation. As a result, each student can see only those *takes* tuples whose ID value matches her ID.

VPD provides authorization at the level of specific tuples, or rows, of a relation, and is therefore said to be a **row-level authorization** mechanism. A potential pitfall with adding a predicate as described above is that it may change the meaning of a query significantly. For example, if a user wrote a query to find the average grade over all courses, she would end up getting the average of *her* grades, not all grades. Although the system would give the "right" answer for the rewritten query, that answer would not correspond to the query the user may have thought she was submitting.

4.8 Summary

- SQL supports several types of joins including natural join, inner and outer joins, and several types of join conditions.

Telegram: @uni_k

www.konkur.in

- Natural join provides a simple way to write queries over multiple relations in which a **where** predicate would otherwise equate attributes with matching names from each relation. This convenience comes at the risk of query semantics changing if a new attribute is added to the schema.
- The **join-using** construct provides a simple way to write queries over multiple relations in which equality is desired for some but not necessarily all attributes with matching names.
- The **join-on** construct provides a way to include a join predicate in the **from** clause.
- Outer join provides a means to retain tuples that, due to a join predicate (whether a natural join, a join-using, or a join-on), would otherwise not appear anywhere in the result relation. The retained tuples are padded with null values so as to conform to the result schema.

- View relations can be defined as relations containing the result of queries. Views are useful for hiding unneeded information and for gathering together information from more than one relation into a single view.
- Transactions are sequences of queries and updates that together carry out a task. Transactions can be committed, or rolled back; when a transaction is rolled back, the effects of all updates performed by the transaction are undone.
- Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.
- Referential-integrity constraints ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- Domain constraints specify the set of possible values that may be associated with an attribute. Such constraints may also prohibit the use of null values for particular attributes.
- Assertions are declarative expressions that state predicates that we require always to be true.
- The SQL data-definition language provides support for defining built-in domain types such as **date** and **time** as well as user-defined domain types.
- Indices are important for efficient processing of queries, as well as for efficient enforcement of integrity constraints. Although not part of the SQL standard, SQL commands for creation of indices are supported by most database systems.
- SQL authorization mechanisms allow one to differentiate among the users of the database on the type of access they are permitted on various data values in the database.

Telegram: @uni_k

www.konkur.in

- Roles enable us to assign a set of privileges to a user according to the role that the user plays in the organization.

Review Terms

- Join types
  - Natural join
  - Inner join with **using** and **on**
  - Left, right and full outer join
  - Outer join with **using** and **on**
- View definition
  - Materialized views
  - View maintenance
  - View update
- Transactions
  - Commit work
  - Rollback work
  - Atomic transaction
- Constraints
  - Integrity constraints
  - Domain constraints
  - Unique constraint
  - Check clause
  - Referential integrity
    - Cascading deletes
    - Cascading updates
  - Assertions
- Data types
  - Date and time types
  - Default values
  - Large objects
    - clob
    - blob
  - User-defined types
  - distinct types
  - Domains
  - Type conversions
- Catalogs
- Schemas
- Indices
- Privileges
  - Types of privileges
    - **select**
    - **insert**
    - **update**
  - Granting of privileges
  - Revoking of privileges
  - Privilege to grant privileges
  - Grant option
- Roles
- Authorization on views
- Execute authorization
- Invoker privileges
- Row-level authorization
- Virtual private database (VPD)

Telegram: @uni_k

www.konkur.in

Practice Exercises

4.1 Consider the following SQL query that seeks to find a list of titles of all courses taught in Spring 2017 along with the name of the instructor.

```
select name, title
from instructor natural join teaches natural join section natural join course
where semester = 'Spring' and year = 2017
```

What is wrong with this query?

4.2 Write the following queries in SQL:

a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

b. Write the same query as in part a, but using a scalar subquery and not using outer join.

c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to "—".

d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

4.3 Outer join expressions can be computed in SQL without using the SQL **outer join** operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the **outer join** expression.

a. **select** * **from** *student* **natural left outer join** *takes*

b. **select** * **from** *student* **natural full outer join** *takes*

4.4 Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**.

a. Give instances of relations r, s, and t such that in the result of
(r **natural left outer join** s) **natural left outer join** t
attribute C has a null value but attribute D has a non-null value.

b. Are there instances of r, s, and t such that the result of
r **natural left outer join** (s **natural left outer join** t)

Telegram: @uni_k

www.konkur.in

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 4.12 Employee database.

has a null value for C but a non-null value for D? Explain why or why not.

4.5 **Testing SQL queries**: To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human checks if the SQL query result on each test database matches the intention of the specification in English.

a. In Section 4.1.1 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result it unintentionally equated the *dept_name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.

b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation for each foreign key. Explain why, using an example query on the university database.

c. When creating test databases, it is important to create tuples with null values for foreign-key attributes, provided the attribute is nullable (SQL allows foreign-key attributes to take on null values, as long as they are not part of the primary key and have not been declared as **not null**). Explain why, using an example query on the university database.

Hint: Use the queries from Exercise 4.2.

4.6 Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; recall that we used a relation *grade_points*(*grade*, *points*) to get the numeric points associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.

4.7 Consider the employee database of Figure 4.12. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Telegram: @uni_k

www.konkur.in

4.8 As discussed in Section 4.4.8, we expect the constraint "an instructor cannot teach sections in two different classrooms in a semester in the same time slot" to hold.

a. Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.

b. Write an SQL assertion to enforce this constraint (as discussed in Section 4.4.8, current generation database systems do not support such assertions, although they are part of the SQL standard).

4.9 SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
    (employee_ID      char(20),
     manager_ID       char(20),
     primary key employee_ID,
     foreign key (manager_ID) references manager(employee_ID)
                                  on delete cascade )
```

Here, *employee_ID* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

4.10 Given the relations *a*(*name, address, title*) and *b*(*name, address, salary*), show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition rather than using the **natural join** syntax. This can be done using the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address* and that the solution is correct even if some tuples in *a* and *b* have null values for attributes *name* or *address*.

4.11 Operating systems usually offer only two types of authorization control for data files: read access and write access. Why do database systems offer so many kinds of authorization?

4.12 Suppose a user wants to grant **select** access on a relation to another user. Why should the user include (or not include) the clause **granted by current role** in the **grant** statement?

4.13 Consider a view *v* whose definition references only relation *r*.

- If a user is granted **select** authorization on *v*, does that user need to have **select** authorization on *r* as well? Why or why not?
- If a user is granted **update** authorization on *v*, does that user need to have **update** authorization on *r* as well? Why or why not?

Telegram: @uni_k

www.konkur.in

- Give an example of an **insert** operation on a view *v* to add a tuple *t* that is not visible in the result of **select** * **from** *v*. Explain your answer.

Exercises

4.14 Consider the query

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2017
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Explain why appending **natural join** *section* in the **from** clause would not change the result.

4.15 Rewrite the query

```
select *
from section natural join classroom
```

without using a natural join but instead using an inner join with a **using** condition.

4.16 Write an SQL query using the university schema to find the ID of each student who has never taken a course at the university. Do this using no subqueries and no set operations (use an outer join).

4.17 Express the following query in SQL using no subqueries and no set operations.

```
select ID
from student
except
select s_id
from advisor
where i_ID is not null
```

4.18 For the database of Figure 4.12, write a query to find the ID of each employee with no manager. Note that an employee may simply have no manager listed or may have a *null* manager. Write your query using an outer join and then write it again using no outer join at all.

4.19 Under what circumstances would the query

Telegram: @uni_k

www.konkur.in


```
select *
from student natural full outer join takes
                    natural full outer join course
```

include tuples with null values for the *title* attribute?

4.20 Show how to define a view *tot_credits* (*year, num_credits*), giving the total number of credits taken in each year.

4.21 For the view of Exercise 4.18, explain why the database system would not allow a tuple to be inserted into the database through this view.

4.22 Show how to express the **coalesce** function using the **case** construct.

4.23 Explain why, when a manager, say Satoshi, grants an authorization, the grant should be done by the manager role, rather than by the user Satoshi.

4.24 Suppose user *A*, who has all authorization privileges on a relation *r*, grants **select** on relation *r* to **public** with grant option. Suppose user *B* then grants **select** on *r* to *A*. Does this cause a cycle in the authorization graph? Explain why.

4.25 Suppose a user creates a new relation *r*1 with a foreign key referencing another relation *r*2. What authorization privilege does the user need on *r*2? Why should this not simply be allowed without any such authorization?

4.26 Explain the difference between integrity constraints and authorization constraints.

Further Reading

General SQL references were provided in Chapter 3. As noted earlier, many systems implement features in a non-standard manner, and, for that reason, a reference specific to the database system you are using is an essential guide. Most vendors also provide extensive support on the web.

The rules used by SQL to determine the updatability of a view, and how updates are reflected on the underlying database relations appeared in SQL:1999 and are summarized in [Melton and Simon (2001)].

The original SQL proposals for assertions date back to [Astrahan et al. (1976)], [Chamberlin et al. (1976)], and [Chamberlin et al. (1981)].

Bibliography

[Astrahan et al. (1976)] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R, A

Telegram: @uni_k

www.konkur.in

Management", *ACM Transactions on Database Systems*, Volume 1, Number 2 (1976), pages 97–137.

[Chamberlin et al. (1976)] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development*, Volume 20, Number 6 (1976), pages 560–575.

[Chamberlin et al. (1981)] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "A History and Evaluation of System R", *Communications of the ACM*, Volume 24, Number 10 (1981), pages 632–646.

[Melton and Simon (2001)] J. Melton and A. R. Simon, *SQL:1999, Understanding Relational Language Components*, Morgan Kaufmann (2001).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 5



Advanced SQL

Chapter 3 and Chapter 4 provided detailed coverage of the basic structure of SQL. In this chapter, we first address the issue of how to access SQL from a general-purpose programming language, which is very important for building applications that use a database to manage data. We then cover some of the more advanced features of SQL, starting with how procedural code can be executed within the database either by extending the SQL language to support procedural actions or by allowing functions defined in procedural languages to be executed within the database. We describe triggers, which can be used to specify actions that are to be carried out automatically on certain events such as insertion, deletion, or update of tuples in a specified relation. Finally, we discuss recursive queries and advanced aggregation features supported by SQL.

5.1 Accessing SQL from a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Python that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.
2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data are only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language.

There are two approaches to accessing SQL from a general-purpose programming language:

Telegram: @uni_k

www.konkur.in

1. **Dynamic SQL**: A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at runtime.

   In this chapter, we look at two standards for connecting to an SQL database and performing queries and updates. One, JDBC (Section 5.1.1), is an application program interface for the Java language. The other, ODBC (Section 5.1.3), is an application program interface originally developed for the C language, and subsequently extended to other languages such as C++, C#, Ruby, Go, PHP, and Visual Basic. We also illustrate how programs written in Python can connect to a database using the Python Database API (Section 5.1.2).

   The ADO.NET API, designed for the Visual Basic .NET and C# languages, provides functions to access data, which at a high level are similar to the JDBC functions, although details differ. The ADO.NET API can also be used with some kinds of non-relational data sources. Details of ADO.NET may be found in the manuals available online and are not covered further in this chapter.

2. **Embedded SQL**: Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor, which translates requests expressed in embedded SQL into function calls. At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities but may be specific to the database that is being used. Section 5.1.4 briefly covers embedded SQL.

A major challenge in mixing SQL with a general-purpose language is the mismatch in the ways these languages manipulate data. In SQL, the primary type of data are relations. SQL statements operate on relations and return relations as a result. Programming languages normally operate on a variable at a time, and those variables correspond roughly to the value of an attribute in a tuple in a relation. Thus, integrating these two types of languages into a single application requires providing a mechanism to return the result of a query in a manner that the program can handle.

Our examples in this section assume that we are accessing a database on a server that runs a database system. An alternative approach using an **embedded database** is discussed in Note 5.1 on page 198.

5.1.1 JDBC

The **JDBC** standard defines an **application program interface (API)** that Java programs can use to connect to database servers. (The word JDBC was originally an abbreviation for **Java Database Connectivity**, but the full form is no longer used.)

Telegram: @uni_k

www.konkur.in

Figure 5.1 shows example Java code that uses the JDBC interface. The Java program must import java.sql.*, which contains the interface definitions for the functionality provided by JDBC.

5.1.1.1 Connecting to the Database

The first step in accessing a database from a Java program is to open a connection to the database. This step is required to select which database to use, such as an instance of Oracle running on your machine, or a PostgreSQL database running on another machine. Only after opening a connection can a Java program execute SQL statements.

```
public static void JDBCexample(String userid, String passwd)
{
    try (
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
    ) {
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987','Kim','Physics',98000)");
        }
        catch (SQLException sqle) {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select dept_name, avg (salary) "+
                " from instructor "+
                " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                    rset.getFloat(2));
        }
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

Figure 5.1 An example of JDBC code.

Telegram: @uni_k

www.konkur.in

A connection is opened using the getConnection() method of the DriverManager class (within java.sql). This method takes three parameters.[1]

1. The first parameter to the getConnection() call is a string that specifies the URL, or machine name, where the server runs (in our example, db.yale.edu), along with possibly some other information such as the protocol to be used to communicate with the database (in our example, jdbc:oracle:thin:; we shall shortly see why this is required), the port number the database system uses for communication (in our example, 2000), and the specific database on the server to be used (in our example, univdb). Note that JDBC specifies only the API, not the communication protocol. A JDBC driver may support multiple protocols, and we must specify one supported by both the database and the driver. The protocol details are vendor specific.
2. The second parameter to getConnection() is a database user identifier, which is a string.
3. The third parameter is a password, which is also a string. (Note that the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.)

In our example in the figure, we have created a Connection object whose handle is conn.

Each database product that supports JDBC (all the major database vendors do) provides a JDBC driver that must be dynamically loaded in order to access the database from Java. In fact, loading the driver must be done first, before connecting to the database. If the appropriate driver has been downloaded from the vendor's web site and is in the classpath, the getConnection() method will locate the needed driver.[2] The driver provides for the translation of product-independent JDBC calls into the product-specific calls needed by the specific database management system being used. The actual protocol used to exchange information with the database depends on the driver that is used, and it is not defined by the JDBC standard. Some drivers support more than one protocol, and a suitable protocol must be chosen depending on what protocol the particular database product supports. In our example, when opening a connection with the database, the string jdbc:oracle:thin: specifies a particular protocol supported by Oracle. The MySQL equivalent is jdbc:mysql:

5.1.1.2 Shipping SQL Statements to the Database System

Once a database connection is open, the program can use it to send SQL statements to the database system for execution. This is done via an instance of the class Statement.

[1]There are multiple versions of the getConnection() method, which differ in the parameters that they accept. We present the most commonly used version.

[2]Prior to version 4, locating the driver was done manually by invoking Class.forName with one argument specifying a concrete class implementing the java.sql.Driver interface, in a line of code

Telegram: @uni_k

www.konkur.in

A Statement object is not the SQL statement itself, but rather an object that allows the Java program to invoke methods that ship an SQL statement given as an argument for execution by the database system. Our example creates a Statement handle (stmt) on the connection conn.

To execute a statement, we invoke either the executeQuery() method or the executeUpdate() method, depending on whether the SQL statement is a query (and, thus, returns a result set) or nonquery statement such as **update**, **insert**, **delete**, or **create table**. In our example, stmt.executeUpdate() executes an update statement that inserts into the *instructor* relation. It returns an integer giving the number of tuples inserted, updated, or deleted. For DDL statements, the return value is zero.

5.1.1.3 Exceptions and Resource Management

Executing any SQL method might result in an exception being thrown. The try { … } catch { … } construct permits us to catch any exceptions (error conditions) that arise when JDBC calls are made and take appropriate action. In JDBC programming, it may be useful to distinguish between an SQLexception, which is an SQL-specific exception, and the general case of an Exception, which could be any Java exception such as a null-pointer exception, or array-index-out-of-bounds exception. We show both in Figure 5.1. In practice, one would write more complete exception handlers than we do (for the sake of conciseness) in our example code.

Opening a connection, a statement, and other JDBC objects are all actions that consume system resources. Programmers must take care to ensure that programs close all such resources. Failure to do so may cause the database system's resource pools to become exhausted, rendering the system inaccessible or inoperative until a time-out period expires. One way to do this is to code explicit calls to close connections and statements. This approach fails if the code exits due to an exception and, in so doing, avoids the Java statement with the close invocation. For this reason, the preferred approach is to use the *try-with-resources* construct in Java. In the example of Figure 5.1, the opening of the connection and statement objects is done within parentheses rather than in the main body of the try in curly braces. Resources opened in the code within parentheses are closed automatically at the end of the try block. This protects us from leaving connections or statements unclosed. Since closing a statement implicitly closes objects opened for that statement (i.e., the ResultSet objects we shall discuss in the next section, this coding practice protects us from leaving resources unclosed.[3] In the example of Figure 5.1, we could have closed the connection explicitly with the statement conn.close() and closed the statement explicitly with stmt.close(), though doing so was not necessary in our example.

5.1.1.4 Retrieving the Result of a Query

The example code of Figure 5.1 executes a query by using stmt.executeQuery(). It retrieves the set of tuples in the result into a ResultSet object rset and fetches them one

[3]This Java feature, called *try-with-resources*, was introduced in Java

Telegram: @uni_k

www.konkur.in

tuple at a time. The next() method on the result set tests whether or not there remains at least one unfetched tuple in the result set and if so, fetches it. The return value of the next() method is a Boolean indicating whether it fetched a tuple. Attributes from the fetched tuple are retrieved using various methods whose names begin with get. The method getString() can retrieve any of the basic SQL data types (converting the value to a Java String object), but more restrictive methods such as getFloat() can be used as well. The argument to the various get methods can either be an attribute name specified as a string, or an integer indicating the position of the desired attribute within the tuple. Figure 5.1 shows two ways of retrieving the values of attributes in a tuple: using the name of the attribute (*dept_name*) and using the position of the attribute (2, to denote the second attribute).

5.1.1.5 Prepared Statements

We can create a prepared statement in which some values are replaced by "?", thereby specifying that actual values will be provided later. The database system compiles the query when it is prepared. Each time the query is executed (with new values to replace the "?"s), the database system can reuse the previously compiled form of the query and apply the new values as parameters. The code fragment in Figure 5.2 shows how prepared statements can be used.

The prepareStatement() method of the Connection class defines a query that may contain parameter values; some JDBC drivers may submit the query to the database for compilation as part of the method, but other drivers do not contact the database at this point. The method returns an object of class PreparedStatement. At this point, no SQL statement has been executed. The executeQuery() and executeUpdate() methods of PreparedStatement class do that. But before they can be invoked, we must use methods of class PreparedStatement that assign values for the "?" parameters. The setString() method and other similar methods such as setInt() for other basic SQL types allow us to specify the values for the parameters. The first argument specifies the "?" parameter for which we are assigning a value (the first parameter is 1, unlike most other Java constructs, which start with 0). The second argument specifies the value to be assigned.

In the example in Figure 5.2, we prepare an **insert** statement, set the "?" parameters, and then invoke executeUpdate(). The final two lines of our example show that parameter assignments remain unchanged until we specifically reassign them. Thus, the final statement, which invokes executeUpdate(), inserts the tuple ("88878", "Perry", "Finance", 125000).

Prepared statements allow for more efficient execution in cases where the same query can be compiled once and then run multiple times with different parameter values. However, there is an even more significant advantage to prepared statements that makes them the preferred method of executing SQL queries whenever a user-entered value is used, even if the query is to be run only once. Suppose that we read in a user-entered value and then use Java string manipulation to construct the statement.

Telegram: @uni_k

www.konkur.in


```
PreparedStatement pStmt = conn.prepareStatement(
                    "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

Figure 5.2 Prepared statements in JDBC code.

If the user enters certain special characters, such as a single quote, the resulting SQL statement may be syntactically incorrect unless we take extraordinary care in checking the input. The setString() method does this for us automatically and inserts the needed escape characters to ensure syntactic correctness.

In our example, suppose that the values for the variables ID, name, dept_name, and salary have been entered by a user, and a corresponding row is to be inserted into the *instructor* relation. Suppose that, instead of using a prepared statement, a query is constructed by concatenating the strings using the following Java expression:

```
"insert into instructor values(' " + ID + " ', ' " + name + " ', " +
                    " '" + dept_name + " ', " + salary + ")"
```

and the query is executed directly using the executeQuery() method of a Statement object. Observe the use of single quotes in the string, which would surround the values of ID, name and dept_name in the generated SQL query.

Now, if the user typed a single quote in the ID or name fields, the query string would have a syntax error. It is quite possible that an instructor name may have a quotation mark in its name (for example, "O'Henry").

While the above example might be considered an annoyance, the situation can be much worse. A technique called **SQL injection** can be used by malicious hackers to steal data or damage the database.

Suppose a Java program inputs a string *name* and constructs the query:

```
"select * from instructor where name = '" + name + "'"
```

If the user, instead of entering a name, enters:

```
X' or 'Y' = 'Y
```

then the resulting statement becomes:

Telegram: @uni_k

www.konkur.in


```
"select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
```

which is:

```
select * from instructor where name = 'X' or 'Y' = 'Y'
```

In the resulting query, the **where** clause is always true and the entire instructor relation is returned.

More clever malicious users could arrange to output even more data, including credentials such as passwords that allow the user to connect to the database and perform any actions they want. SQL injection attacks on **update** statements can be used to change the values that are being stored in updated columns. In fact there have been a number of attacks in the real world using SQL injections; attacks on multiple financial sites have resulted in theft of large amounts of money by using SQL injection attacks.

Use of a prepared statement would prevent this problem because the input string would have escape characters inserted, so the resulting query becomes:

```
"select * from instructor where name = 'X\' or \'Y\' = \'Y'
```

which is harmless and returns the empty relation.

Programmers must pass user-input strings to the database only through parameters of prepared statements; creating SQL queries by concatenating strings with user-input values is an extremely serious security risk and should never be done in any program.

Some database systems allow multiple SQL statements to be executed in a single JDBC `execute` method, with statements separated by a semicolon. This feature has been turned off by default on some JDBC drivers because it allows malicious hackers to insert whole SQL statements using SQL injection. For instance, in our earlier SQL injection example a malicious user could enter:

```
X'; drop table instructor; - -
```

which will result in a query string with two statements separated by a semicolon being submitted to the database. Because these statements run with the privileges of the database userid used by the JDBC connection, devastating SQL statements such as **drop table**, or updates to any table of the user's choice, could be executed. However, some databases still allow execution of multiple statements as above; it is thus very important to correctly use prepared statements to avoid the risk of SQL injection.

5.1.1.6 Callable Statements

JDBC also provides a `CallableStatement` interface that allows invocation of SQL stored procedures and functions (described in Section 5.2). These play the same role for functions and procedures as `prepareStatement` does for queries.

Telegram: @uni_k

www.konkur.in


```
CallableStatement cStmt1 = conn.prepareCall("{? = call some_function(?)}");
CallableStatement cStmt2 = conn.prepareCall("{call some_procedure(?,?)}");
```

The data types of function return values and out parameters of procedures must be registered using the method registerOutParameter(), and can be retrieved using get methods similar to those for result sets. See a JDBC manual for more details.

5.1.1.7 Metadata Features

As we noted earlier, a Java application program does not include declarations for data stored in the database. Those declarations are part of the SQL DDL statements. Therefore, a Java program that uses JDBC must either have assumptions about the database schema hard-coded into the program or determine that information directly from the database system at runtime. The latter approach is usually preferable, since it makes the application program more robust to changes in the database schema.

Recall that when we submit a query using the executeQuery() method, the result of the query is contained in a ResultSet object. The interface ResultSet has a method, getMetaData(), that returns a ResultSetMetaData object that contains metadata about the result set. ResultSetMetaData, in turn, has methods to find metadata information, such as the number of columns in the result, the name of a specified column, or the type of a specified column. In this way, we can write code to execute a query even if we have no prior knowledge of the schema of the result.

The following Java code segment uses JDBC to print out the names and types of all columns of a result set. The variable rs in the code is assumed to refer to a ResultSet instance obtained by executing a query.

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
      System.out.println(rsmd.getColumnName(i));
      System.out.println(rsmd.getColumnTypeName(i));
}
```

The getColumnCount() method returns the arity (number of attributes) of the result relation. That allows us to iterate through each attribute (note that we start at 1, as is conventional in JDBC). For each attribute, we retrieve its name and data type using the methods getColumnName() and getColumnTypeName(), respectively.

The DatabaseMetaData interface provides a way to find metadata about the database. The interface Connection has a method getMetaData() that returns a DatabaseMetaData object. The DatabaseMetaData interface in turn has a very large number of methods to get metadata about the database and the database system to which the application is connected.

For example, there are methods that return the product name and version number of the database system. Other methods allow the application to query the database system about its supported features.

Telegram: @uni_k

www.konkur.in


```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
    // Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
    //          and Column-Pattern
    // Returns: One row for each column; row has a number of attributes
    //          such as COLUMN_NAME, TYPE_NAME
while( rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"),
            rs.getString("TYPE_NAME");
}
```

Figure 5.3 Finding column information in JDBC using DatabaseMetaData.

Still other methods return information about the database itself. The code in Figure 5.3 illustrates how to find information about columns (attributes) of relations in a database. The variable conn is assumed to be a handle for an already opened database connection. The method getColumns() takes four arguments: a catalog name (null signifies that the catalog name is to be ignored), a schema name pattern, a table name pattern, and a column name pattern. The schema name, table name, and column name patterns can be used to specify a name or a pattern. Patterns can use the SQL string matching special characters "%" and "_"; for instance, the pattern "%" matches all names. Only columns of tables of schemas satisfying the specified name or pattern are retrieved. Each row in the result set contains information about one column. The rows have a number of columns such as the name of the catalog, schema, table and column, the type of the column, and so on.

The getTables() method allows you to get a list of all tables in the database. The first three parameters to getTables() are the same as for getColumns(). The fourth parameter can be used to restrict the types of tables returned; if set to null, all tables, including system internal tables are returned, but the parameter can be set to restrict the tables returned to only user-created tables.

Examples of other methods provided by DatabaseMetaData that provide information about the database include those for primary keys (getPrimaryKeys()), foreign-key references (getCrossReference()), authorizations, database limits such as maximum number of connections, and so on.

The metadata interfaces can be used for a variety of tasks. For example, they can be used to write a database browser that allows a user to find the tables in a database, examine their schema, examine rows in a table, apply selections to see desired rows, and so on. The metadata information can be used to make code used for these tasks generic; for example, code to display the rows in a relation can be written in such a way that it would work on all possible relations regardless of their schema. Similarly, it is

Telegram: @uni_k

www.konkur.in

possible to write code that takes a query string, executes the query, and prints out the results as a formatted table; the code can work regardless of the actual query submitted.

5.1.1.8 Other Features

JDBC provides a number of other features, such as **updatable result sets**. It can create an updatable result set from a query that performs a selection and/or a projection on a database relation. An update to a tuple in the result set then results in an update to the corresponding tuple of the database relation.

Recall from Section 4.3 that a transaction allows multiple actions to be treated as a single atomic unit which can be committed or rolled back. By default, each SQL statement is treated as a separate transaction that is committed automatically. The method setAutoCommit() in the JDBC Connection interface allows this behavior to be turned on or off. Thus, if conn is an open connection, conn.setAutoCommit(false) turns off automatic commit. Transactions must then be committed or rolled back explicitly using either conn.commit() or conn.rollback(). conn.setAutoCommit(true) turns on automatic commit.

JDBC provides interfaces to deal with large objects without requiring an entire large object to be created in memory. To fetch large objects, the ResultSet interface provides methods getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively. These objects do not store the entire large object, but instead store "locators" for the large objects, that is, logical pointers to the actual large object in the database. Fetching data from these objects is very much like fetching data from a file or an input stream, and it can be performed using methods such as getBytes() and getSubString().

Conversely, to store large objects in the database, the PreparedStatement class permits a database column whose type is **blob** to be linked to an input stream (such as a file that has been opened) using the method setBlob(int parameterIndex, InputStream inputStream). When the prepared statement is executed, data are read from the input stream and written to the **blob** in the database. Similarly, a **clob** column can be set using the setClob() method, which takes as arguments a parameter index and a character stream.

JDBC includes a *row set* feature that allows result sets to be collected and shipped to other applications. Row sets can be scanned both backward and forward and can be modified.

5.1.2 Database Access from Python

Database access can be done from Python as illustrated by the method shown in Figure 5.4. The statement containing the insert query shows how to use the Python equivalent of JDBC prepared statements, with parameters identified in the SQL query by "%s", and parameter values provided as a list. Updates are not committed to the database automatically; the commit() method needs to be called to commit an update.

Telegram: @uni_k

www.konkur.in


```
import psycopg2

def PythonDatabaseExample(userid, passwd)
    try:
        conn = psycopg2.connect( host="db.yale.edu", port=5432,
                dbname="univdb", user=userid, password=passwd)
        cur = conn.cursor()
        try:
            cur.execute("insert into instructor values(%s, %s, %s, %s)",
                    ("77987","Kim","Physics",98000))
            conn.commit();
        except Exception as sqle:
            print("Could not insert tuple. ", sqle)
            conn.rollback()
        cur.execute( ("select dept_name, avg (salary) "
                    " from instructor group by dept_name"))
        for dept in cur:
            print dept[0], dept[1]
    except Exception as sqle:
        print("Exception : ", sqle)
```

Figure 5.4 Database access from Python

The try:, except …: block shows how to catch exceptions and to print information about the exception. The for loop illustrates how to loop over the result of a query execution, and to access individual attributes of a particular row.

The preceding program uses the psycopg2 driver, which allows connection to PostgreSQL databases and is imported in the first line of the program. Drivers are usually database specific, with the MySQLdb driver to connect to MySQL, and cx_Oracle to connect to Oracle; but the pyodbc driver can connect to most databases that support ODBC. The Python Database API used in the program is implemented by drivers for many databases, but unlike with JDBC, there are minor differences in the API across different drivers, in particular in the parameters to the connect() function.

5.1.3 ODBC

The **Open Database Connectivity** (**ODBC**) standard defines an API that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports ODBC.

Each database system supporting ODBC provides a library that must be linked with the client program. When the client program makes an call, the code

Telegram: @uni_k

www.konkur.in


```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
    {
        char deptname[80];
        float salary;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * sqlquery = "select dept_name, sum (salary)
                           from instructor
                           group by dept_name";
        SQLAllocStmt(conn, &stmt);
        error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, deptname , 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
            while (SQLFetch(stmt) == SQL_SUCCESS) {
                printf (" %s %g\n", deptname, salary);
            }
        }
        SQLFreeStmt(stmt, SQL_DROP);
    }
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Figure 5.5 ODBC code example.

in the library communicates with the server to carry out the requested action and fetch results.

Figure 5.5 shows an example of C code using the ODBC API. The first step in using ODBC to communicate with a server is to set up a connection with the server. To do so, the program first allocates an SQL environment, then a database connection handle. ODBC defines the types HENV, HDBC, and . The program then opens the

Telegram: @uni_k

www.konkur.in

database connection by using SQLConnect. This call takes several parameters, including the connection handle, the server to which to connect, the user identifier, and the password for the database. The constant SQL_NTS denotes that the previous argument is a null-terminated string.

Once the connection is set up, the program can send SQL commands to the database by using SQLExecDirect. C language variables can be bound to attributes of the query result, so that when a result tuple is fetched using SQLFetch, its attribute values are stored in corresponding C variables. The SQLBindCol function does this task; the second argument identifies the position of the attribute in the query result, and the third argument indicates the type conversion required from SQL to C. The next argument gives the address of the variable. For variable-length types like character arrays, the last two arguments give the maximum length of the variable and a location where the actual length is to be stored when a tuple is fetched. A negative value returned for the length field indicates that the value is **null**. For fixed-length types such as integer or float, the maximum length field is ignored, while a negative value returned for the length field indicates a null value.

The SQLFetch statement is in a **while** loop that is executed until SQLFetch returns a value other than SQL_SUCCESS. On each fetch, the program stores the values in C variables as specified by the calls on SQLBindCol and prints out these values.

At the end of the session, the program frees the statement handle, disconnects from the database, and frees up the connection and SQL environment handles. Good programming style requires that the result of every function call must be checked to make sure there are no errors; we have omitted most of these checks for brevity.

It is possible to create an SQL statement with parameters; for example, consider the statement insert into department values(?,?,?). The question marks are placeholders for values which will be supplied later. The above statement can be "prepared," that is, compiled at the database, and repeatedly executed by providing actual values for the placeholders—in this case, by providing a department name, building, and budget for the relation *department*.

ODBC defines functions for a variety of tasks, such as finding all the relations in the database and finding the names and types of columns of a query result or a relation in the database.

By default, each SQL statement is treated as a separate transaction that is committed automatically. The SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0) turns off automatic commit on connection conn, and transactions must then be committed explicitly by SQLTransact(conn, SQL_COMMIT) or rolled back by SQLTransact(conn, SQL_ROLLBACK).

The ODBC standard defines *conformance levels*, which specify subsets of the functionality defined by the standard. An ODBC implementation may provide only core level features, or it may provide more advanced (level 1 or level 2) features. Level 1 requires support for fetching information about the catalog, such as information about what relations are present and the types of their attributes. Level 2 requires further fea-

Telegram: @uni_k

www.konkur.in

tures, such as the ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

The SQL standard defines a **call level interface (CLI)** that is similar to the ODBC interface.

5.1.4 Embedded SQL

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, C++, Cobol, Pascal, Java, PL/I, and Fortran. A language in which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language constitute *embedded* SQL.

Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database. An embedded SQL program must be processed by a special preprocessor prior to compilation. The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow runtime execution of the database accesses. Then the resulting program is compiled by the host-language compiler. This is the main distinction between embedded SQL and JDBC or ODBC.

To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement; it has the form:

EXEC SQL <embedded SQL statement >;

Before executing any SQL statements, the program must first connect to the database. Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

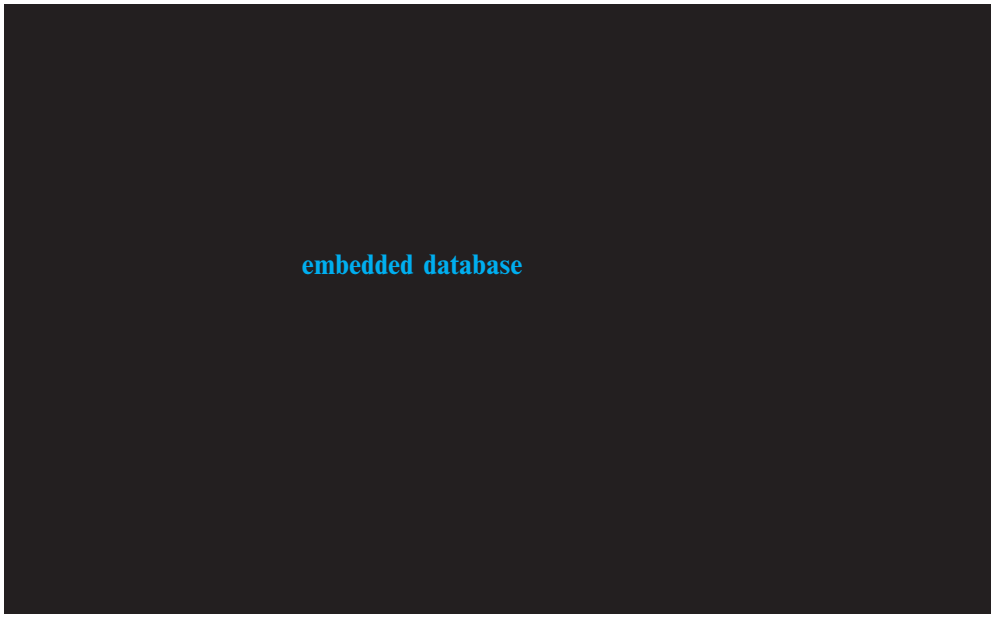
To iterate over the results of an embedded SQL query, we must declare a *cursor* variable, which can then be opened, and *fetch* commands issued in a host language loop to fetch consecutive rows of the query result. Attributes of a row can be fetched into host language variables. Database updates can also be performed using a cursor on a relation to iterate through the rows of the relation, optionally using a **where** clause to iterate through only selected rows. Embedded SQL commands can be used to update the current row where the cursor is pointing.

The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. You may refer to the manuals of the specific language embedding that you use for further details.

In JDBC, SQL statements are interpreted at runtime (even if they are created using the prepared statement feature). When embedded SQL is used, there is a potential for catching some SQL-related errors (including data-type errors) at the time of preprocessing. SQL queries in embedded SQL programs are also easier to comprehend than in programs using dynamic SQL. However, there are also some disadvantages with embedded SQL. The preprocessor creates new host language code, which may complicate debugging of the program. The constructs used by the preprocessor to identify

Telegram: @uni_k

www.konkur.in



statements may clash syntactically with host language syntax introduced in subsequent versions of the host language.

As a result, most current systems use dynamic SQL, rather than embedded SQL. One exception is the Microsoft Language Integrated Query (LINQ) facility, which extends the host language to include support for queries instead of using a preprocessor to translate embedded SQL queries into the host language.

5.2 Functions and Procedures

We have already seen several functions that are built into the SQL language. In this section, we show how developers can write their own functions and procedures, store them in the database, and then invoke them from SQL statements. Functions are particularly useful with specialized data types such as images and geometric objects. For instance, a line-segment data type used in a map database may have an associated function that checks whether two line segments overlap, and an image data type may have associated functions to compare two images for similarity.

Procedures and functions allow "business logic" to be stored in the database and executed from SQL statements. For example, universities usually have many rules about how many courses a student can take in a given semester, the minimum number of courses a full-time instructor must teach in a year, the maximum number of majors a student can be enrolled in, and so on. While such business logic can be encoded as programming-language procedures stored entirely outside the database, defining them as stored procedures in the database has several advantages. For example, it allows

Telegram: @uni_k

www.konkur.in


```
create function dept_count(dept_name varchar(20))
    returns integer
    begin
    declare d_count integer;
        select count( *) into d_count
        from instructor
        where instructor.dept_name= dept_name
    return d_count;
    end
```

Figure 5.6 Function defined in SQL.

multiple applications to access the procedures, and it allows a single point of change in case the business rules change, without changing other parts of the application. Application code can then call the stored procedures instead of directly updating database relations.

SQL allows the definition of functions, procedures, and methods. These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++. We look at definitions in SQL first and then see how to use definitions in external languages in Section 5.2.3.

Although the syntax we present here is defined by the SQL standard, most databases implement nonstandard versions of this syntax. For example, the procedural languages supported by Oracle (PL/SQL), Microsoft SQL Server (TransactSQL), and PostgreSQL (PL/pgSQL) all differ from the standard syntax we present here. We illustrate some of the differences for the case of Oracle in Note 5.2 on page 204. See the respective system manuals for further details. Although parts of the syntax we present here may not be supported on such systems, the concepts we describe are applicable across implementations, although with a different syntax.

5.2.1 Declaring and Invoking SQL Functions and Procedures

Suppose that we want a function that, given the name of a department, returns the count of the number of instructors in that department. We can define the function as shown in Figure 5.6.[4] This function can be used in a query that returns names and budgets of all departments with more than 12 instructors:

```
select dept_name, budget
from department
where dept_count(dept_name) > 12;
```

[4] If you are entering your own functions or procedures, you should write "**create or replace**" rather than **create** so that it is easy to modify your code (by replacing the function) during

Telegram: @uni_k

www.konkur.in


```
create function instructor_of (dept_name varchar(20))
    returns table (
        ID varchar (5),
        name varchar (20),
        dept_name varchar (20),
        salary numeric (8,2))
return table
    (select ID, name, dept_name, salary
    from instructor
    where instructor.dept_name = instructor_of.dept_name);
```

Figure 5.7 Table function in SQL.

Performance problems have been observed on many database systems when invoking complex user-defined functions within a query, if the functions are invoked on a large number of tuples. Programmers should therefore take performance into consideration when deciding whether to use user-defined functions in a query.

The SQL standard supports functions that can return tables as results; such functions are called **table functions**. Consider the function defined in Figure 5.7. The function returns a table containing all the instructors of a particular department. Note that the function's parameter is referenced by prefixing it with the name of the function (*instructor_of.dept_name*).

The function can be used in a query as follows:

```
select *
from table(instructor_of ('Finance'));
```

This query returns all instructors of the 'Finance' department. In this simple case it is straightforward to write this query without using table-valued functions. In general, however, table-valued functions can be thought of as **parameterized views** that generalize the regular notion of views by allowing parameters.

SQL also supports procedures. The *dept_count* function could instead be written as a procedure:

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out d_count integer)
begin
    select count( *) into d_count
    from instructor
    where instructor.dept_name= dept_count_proc.dept_name
end
```

Telegram: @uni_k

www.konkur.in

The keywords **in** and **out** indicate, respectively, parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.

Procedures can be invoked either from an SQL procedure or from embedded SQL by the **call** statement:

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

Procedures and functions can be invoked from dynamic SQL, as illustrated by the JDBC syntax in Section 5.1.1.5.

SQL permits more than one procedure of the same name, so long as the number of arguments of the procedures with the same name is different. The name, along with the number of arguments, is used to identify the procedure. SQL also permits more than one function with the same name, so long as the different functions with the same name either have different numbers of arguments, or for functions with the same number of arguments, they differ in the type of at least one argument.

5.2.2 Language Constructs for Procedures and Functions

SQL supports constructs that give it almost all the power of a general-purpose programming language. The part of the SQL standard that deals with these constructs is called the **Persistent Storage Module (PSM)**.

Variables are declared using a **declare** statement and can have any valid SQL data type. Assignments are performed using a **set** statement.

A compound statement is of the form **begin** … **end**, and it may contain multiple SQL statements between the **begin** and the **end**. Local variables can be declared within a compound statement, as we have seen in Section 5.2.1. A compound statement of the form **begin atomic** … **end** ensures that all the statements contained within it are executed as a single transaction.

The syntax for **while** statements and **repeat** statements is:

```
while boolean expression do
    sequence of statements;
end while

repeat
    sequence of statements;
until boolean expression
end repeat
```

There is also a **for** loop that permits iteration over all the results of a query:

Telegram: @uni_k

www.konkur.in


```
declare n integer default 0;
for r as
        select budget from department
        where dept_name = 'Music'
do
    set n = n− r.budget
end for
```

The program fetches the query results one row at a time into the **for** loop variable (*r*, in the above example). The statement **leave** can be used to exit the loop, while **iterate** starts on the next tuple, from the beginning of the loop, skipping the remaining statements.

The conditional statements supported by SQL include **if-then-else** statements by using this syntax:

```
if boolean expression
    then statement or compound statement
elseif boolean expression
    then statement or compound statement
else statement or compound statement
end if
```

SQL also supports a case statement similar to the C/C++ language case statement (in addition to case expressions, which we saw in Chapter 3).

Figure 5.8 provides a larger example of the use of procedural constructs in SQL. The function *registerStudent* defined in the figure registers a student in a course section after verifying that the number of students in the section does not exceed the capacity of the room allocated to the section. The function returns an error code—a value greater than or equal to 0 signifies success, and a negative value signifies an error condition—and a message indicating the reason for the failure is returned as an **out** parameter.

The SQL procedural language also supports the signaling of **exception conditions** and declaring of **handlers** that can handle the exception, as in this code:

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
sequence of statements
end
```

The statements between the **begin** and the **end** can raise an exception by executing **signal** *out_of_classroom_seats*. The handler says that if the condition arises, the action to be taken is to exit the enclosing **begin end** statement. Alternative actions would be **continue**, which continues execution from the next statement following the one that raised the exception. In addition to explicitly defined conditions, there are also predefined conditions such as **sqlexception**, **sqlwarning**, and **not**

Telegram: @uni_k

www.konkur.in


```
- - Registers a student after ensuring classroom capacity is not exceeded
- - Returns 0 on success, and -1 if capacity is exceeded.
create function registerStudent(
                    in s_id varchar(5),
                    in s_courseid varchar (8),
                    in s_secid varchar (8),
                    in s_semester varchar (6),
                    in s_year numeric (4,0),
                    out errorMsg varchar(100)
returns integer
begin
    declare currEnrol int;
    select count(*) into currEnrol
        from takes
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
        from classroom natural join section
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    if (currEnrol < limit)
        begin
            insert into takes values
                (s_id, s_courseid, s_secid, s_semester, s_year, null);
            return(0);
        end
    - - Otherwise, section capacity limit already reached
    set errorMsg = 'Enrollment limit reached for course ' || s_courseid
        || ' section ' || s_secid;
    return(-1);
end;
```

Figure 5.8 Procedure to register a student for a course section.

5.2.3 External Language Routines

Although the procedural extensions to SQL can be very useful, they are unfortunately not supported in a standard way across databases. Even the most basic features have different syntax or semantics in different database products. As a result, programmers have to learn a new language for each database product. An alternative that is gaining

Telegram: @uni_k

www.konkur.in



support is to define procedures in an imperative programming language, but allow them to be invoked from SQL queries and trigger definitions.

SQL allows us to define functions in a programming language such as Java, C#, C, or C++. Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.

Telegram: @uni_k

www.konkur.in

External procedures and functions can be specified in this way (note that the exact syntax depends on the specific database system you use):

```
create procedure dept_count_proc( in dept_name varchar(20),
                                  out count integer)
language C
external name '/usr/avi/bin/dept_count_proc'

create function dept_count (dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

In general, the external language procedures need to deal with null values in parameters (both **in** and **out**) and return values. They also need to communicate failure/success status and to deal with exceptions. This information can be communicated by extra parameters: an **sqlstate** value to indicate failure/success status, a parameter to store the return value of the function, and indicator variables for each parameter/function result to indicate if the value is null. Other mechanisms are possible to handle null values, for example, by passing pointers instead of values. The exact mechanisms depend on the database. However, if a function does not deal with these situations, an extra line **parameter style general** can be added to the declaration to indicate that the external procedures/functions take only the arguments shown and do not handle null values or exceptions.

Functions defined in a programming language and compiled outside the database system may be loaded and executed with the database-system code. However, doing so carries the risk that a bug in the program can corrupt the internal structures of the database and can bypass the access-control functionality of the database system. Database systems that are concerned more about efficient performance than about security may execute procedures in such a fashion. Database systems that are concerned about security may execute such code as part of a separate process, communicate the parameter values to it, and fetch results back via interprocess communication. However, the time overhead of interprocess communication is quite high; on typical CPU architectures, tens to hundreds of thousands of instructions can execute in the time taken for one interprocess communication.

If the code is written in a "safe" language such as Java or C#, there is another possibility: executing the code in a **sandbox** within the database query execution process itself. The sandbox allows the Java or C# code to access its own memory area, but it prevents the code from reading or updating the memory of the query execution process, or accessing files in the file system. (Creating a sandbox is not possible for a language such as C, which allows unrestricted access to memory through pointers.) Avoiding interprocess communication reduces function call overhead greatly.

Telegram: @uni_k

www.konkur.in

Several database systems today support external language routines running in a sandbox within the query execution process. For example, Oracle and IBM DB2 allow Java functions to run as part of the database process. Microsoft SQL Server allows procedures compiled into the Common Language Runtime (CLR) to execute within the database process; such procedures could have been written, for example, in C# or Visual Basic. PostgreSQL allows functions defined in several languages, such as Perl, Python, and Tcl.

5.3 Triggers

A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database. To define a trigger, we must:

- Specify when a trigger is to be executed. This is broken up into an *event* that causes the trigger to be checked and a *condition* that must be satisfied for trigger execution to proceed.
- Specify the *actions* to be taken when the trigger executes.

Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition is satisfied.

5.3.1 Need for Triggers

Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL. Triggers are also useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met. As an illustration, we could design a trigger that, whenever a tuple is inserted into the *takes* relation, updates the tuple in the *student* relation for the student taking the course by adding the number of credits for the course to the student's total credits. As another example, suppose a warehouse wishes to maintain a minimum inventory of each item; when the inventory level of an item falls below the minimum level, an order can be placed automatically. On an update of the inventory level of an item, the trigger compares the current inventory level with the minimum inventory level for the item, and if the level is at or below the minimum, a new order is created.

Note that triggers cannot usually perform updates outside the database, and hence, in the inventory replenishment example, we cannot use a trigger to place an order in the external world. Instead, we add an order to a relation holding reorders. We must create a separate permanently running system process that periodically scans that relation and places orders. Some database systems provide built-in support for sending email from SQL queries and triggers using this approach.

Telegram: @uni_k

www.konkur.in


```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
        select time_slot_id
        from time_slot)) /* time_slot_id not present in time_slot */
begin
  rollback
end;

create trigger timeslot_check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
        select time_slot_id
        from time_slot) /* last tuple for time_slot_id deleted from time_slot */
  and orow.time_slot_id in (
        select time_slot_id
        from section)) /* and time_slot_id still referenced from section*/
begin
  rollback
end;
```

Figure 5.9 Using triggers to maintain referential integrity.

5.3.2 Triggers in SQL

We now consider how to implement triggers in SQL. The syntax we present here is defined by the SQL standard, but most databases implement nonstandard versions of this syntax. Although the syntax we present here may not be supported on such systems, the concepts we describe are applicable across implementations. We discuss nonstandard trigger implementations in Note 5.3 on page 212. In each system, trigger syntax is based upon that system's syntax for coding functions and procedures.

Figure 5.9 shows how triggers can be used to ensure referential integrity on the *time_slot_id* attribute of the *section* relation. The first trigger definition in the figure specifies that the trigger is initiated *after* any insert on the relation *section* and it ensures that the *time_slot_id* value being inserted is valid. SQL bf insert statement could insert multiple tuples of the relation, and the **for each row** clause in the trigger code would then explicitly iterate over each inserted row. The **referencing new row as** clause creates a variable *nrow* (called a **transition variable**) that stores the value of the row being inserted.

Telegram: @uni_k

www.konkur.in

The **when** statement specifies a condition. The system executes the rest of the trigger body only for tuples that satisfy the condition. The **begin atomic** … **end** clause can serve to collect multiple SQL statements into a single compound statement. In our example, though, there is only one statement, which rolls back the transaction that caused the trigger to get executed. Thus, any transaction that violates the referential integrity constraint gets rolled back, ensuring the data in the database satisfies the constraint.

It is not sufficient to check referential integrity on inserts alone; we also need to consider updates of *section*, as well as deletes and updates to the referenced table *time_slot*. The second trigger definition in Figure 5.9 considers the case of deletes to *time_slot*. This trigger checks that the *time_slot_id* of the tuple being deleted is either still present in *time_slot*, or that no tuple in *section* contains that particular *time_slot_id* value; otherwise, referential integrity would be violated.

To ensure referential integrity, we would also have to create triggers to handle updates to *section* and *time_slot*; we describe next how triggers can be executed on updates, but we leave the definition of these triggers as an exercise to the reader.

For updates, the trigger can specify attributes whose update causes the trigger to execute; updates to other attributes would not cause it to be executed. For example, to specify that a trigger executes after an update to the *grade* attribute of the *takes* relation, we write:

after update of *takes* **on** *grade*

The **referencing old row as** clause can be used to create a variable storing the old value of an updated or deleted row. The **referencing new row as** clause can be used with updates in addition to inserts.

Figure 5.10 shows how a trigger can be used to keep the *tot_cred* attribute value of *student* tuples up-to-date when the *grade* attribute is updated for a tuple in the *takes* relation. The trigger is executed only when the *grade* attribute is updated from a value that is either null or 'F' to a grade that indicates the course is successfully completed. The **update** statement is normal SQL syntax except for the use of the variable *nrow*.

A more realistic implementation of this example trigger would also handle grade corrections that change a successful completion grade to a failing grade and handle insertions into the *takes* relation where the *grade* indicates successful completion. We leave these as an exercise for the reader.

As another example of the use of a trigger, the action on **delete** of a *student* tuple could be to check if the student has any entries in the *takes* relation, and if so, to delete them.

Many database systems support a variety of other triggering events, such as when a user (application) logs on to the database (that is, opens a connection), the system shuts down, or changes are made to system settings.

Triggers can be activated **before** the event (**insert**, **delete**, or **update**) instead of **after** the event. Triggers that execute before an event can serve as extra constraints that can prevent invalid updates, inserts, or deletes. Instead of letting the invalid action proceed

Telegram: @uni_k

www.konkur.in


```
create trigger credits_earned after update of takes on grade
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
   update student
   set tot_cred= tot_cred+
            (select credits
             from course
             where course.course_id= nrow.course_id)
   where student.id = nrow.id;
end;
```

Figure 5.10 Using a trigger to maintain *credits_earned* values.

and cause an error, the trigger might take action to correct the problem so that the **update**, **insert**, or **delete** becomes valid. For example, if we attempt to insert an instructor into a department whose name does not appear in the *department* relation, the trigger could insert a tuple into the *department* relation for that department name before the insertion generates a foreign-key violation. As another example, suppose the value of an inserted grade is blank, presumably to indicate the absence of a grade. We can define a trigger that replaces the value with the **null** value. The **set** statement can be used to carry out such modifications. An example of such a trigger appears in Figure 5.11.

Instead of carrying out an action for each affected row, we can carry out a single action for the entire SQL statement that caused the insert, delete, or update. To do so, we use the **for each statement** clause instead of the **for each row** clause. The clauses

```
create trigger setnull before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
   set nrow.grade = null;
end;
```

Figure 5.11 Example of using **set** to change an inserted value.

Telegram: @uni_k

www.konkur.in

referencing old table as or **referencing new table as** can then be used to refer to temporary tables (called *transition tables*) containing all the affected rows. Transition tables cannot be used with **before** triggers, but they can be used with **after** triggers, regardless of whether they are statement triggers or row triggers. A single SQL statement can then be used to carry out multiple actions on the basis of the transition tables.

Triggers can be disabled or enabled; by default they are enabled when they are created, but they can be disabled by using **alter trigger** *trigger_name* **disable** (some databases use alternative syntax such as **disable trigger** *trigger_name*). A trigger that has been disabled can be enabled again. A trigger can instead be dropped, which removes it permanently, by using the command **drop trigger** *trigger_name*.

Returning to our inventory-replenishment example from Section 5.3.1, suppose we have the following relations:

- *inventory* (*item, level*), which notes the current amount of the item in the warehouse.
- *minlevel* (*item, level*), which notes the minimum amount of the item to be maintained.
- *reorder* (*item, amount*), which notes the amount of the item to be ordered when its level falls below the minimum.
- *orders* (*item, amount*), which notes the amount of the item to be ordered.

To place a reorder when inventory falls below a specified minimum, we can use the trigger shown in Figure 5.12. Note that we have been careful to place an order only when the amount falls from above the minimum level to below the minimum level. If we check only that the new value after an update is below the minimum level, we may place an order erroneously when the item has already been reordered.

SQL-based database systems use triggers widely, although before SQL:1999 they were not part of the SQL standard. Unfortunately, as a result, each database system implemented its own syntax for triggers, leading to incompatibilities. The SQL:1999 syntax for triggers that we use here is similar, but not identical, to the syntax in the IBM DB2 and Oracle database systems. See Note 5.3 on page 212.

5.3.3 When Not to Use Triggers

There are many good uses for triggers, such as those we have just seen in Section 5.3.2, but some uses are best handled by alternative techniques. For example, we could implement the **on delete cascade** feature of a foreign-key constraint by using a trigger instead of using the cascade feature. Not only would this be more work to implement, but also it would be much harder for a database user to understand the set of constraints implemented in the database.

Telegram: @uni_k

www.konkur.in


```
create trigger reorder after update of level on inventory
referencing old row as orow, new row as nrow
for each row
when nrow.level <= (select level
                    from minlevel
                    where minlevel.item = orow.item)
and orow.level > (select level
                  from minlevel
                  where minlevel.item = orow.item)
begin atomic
    insert into orders
        (select item, amount
         from reorder
         where reorder.item = orow.item);
end;
```

Figure 5.12 Example of trigger for reordering an item.

As another example, triggers can be used to maintain materialized views. For instance, if we wished to support very fast access to the total number of students registered for each course section, we could do this by creating a relation

section_registration(*course_id*, *sec_id*, *semester*, *year*, *total_students*)

defined by the query

```
select course_id, sec_id, semester, year, count(ID) as total_students
from takes
group by course_id, sec_id, semester, year;
```

The value of *total_students* for each course must be maintained up-to-date by triggers on insert, delete, or update of the *takes* relation. Such maintenance may require insertion, update or deletion of tuples from *section_registration*, and triggers must be written accordingly.

However, many database systems now support materialized views, which are automatically maintained by the database system (see Section 4.2.3). As a result, there is no need to write trigger code for maintaining such materialized views.

Triggers have been used for maintaining copies, or replicas, of databases. A collection of triggers on insert, delete, or update can be created on each relation to record the changes in relations called **change** or **delta** relations. A separate process copies over the changes to the replica of the database. Modern database systems, however, provide

Telegram: @uni_k

www.konkur.in



built-in facilities for database replication, making triggers unnecessary for replication in most cases. Replicated databases are discussed in detail in Chapter 23.

Another problem with triggers lies in unintended execution of the triggered action when data are loaded from a backup copy,[5] or when database updates at a site are replicated on a backup site. In such cases, the triggered action has already been executed, and typically it should not be executed again. When loading data, triggers can be disabled explicitly. For backup replica systems that may have to take over from the primary system, triggers would have to be disabled initially and enabled when the backup

[5]We discuss database backup and recovery from failures in detail in Chapter

Telegram: @uni_k

www.konkur.in

BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Figure 5.13 An instance of the *prereq* relation.

site takes over processing from the primary system. As an alternative, some database systems allow triggers to be specified as **not for replication**, which ensures that they are not executed on the backup site during database replication. Other database systems provide a system variable that denotes that the database is a replica on which database actions are being replayed; the trigger body should check this variable and exit if it is true. Both solutions remove the need for explicit disabling and enabling of triggers.

Triggers should be written with great care, since a trigger error detected at runtime causes the failure of the action statement that set off the trigger. Furthermore, the action of one trigger can set off another trigger. In the worst case, this could even lead to an infinite chain of triggering. For example, suppose an insert trigger on a relation has an action that causes another (new) insert on the same relation. The insert action then triggers yet another insert action, and so on ad infinitum. Some database systems limit the length of such chains of triggers (for example, to 16 or 32) and consider longer chains of triggering an error. Other systems flag as an error any trigger that attempts to reference the relation whose modification caused the trigger to execute in the first place.

Triggers can serve a very useful purpose, but they are best avoided when alternatives exist. Many trigger applications can be substituted by appropriate use of stored procedures, which we discussed in Section 5.2.

5.4 Recursive Queries

Consider the instance of the relation *prereq* shown in Figure 5.13 containing information about the various courses offered at the university and the prerequisite for each course.[6]

Suppose now that we want to find out which courses are a prerequisite whether directly or indirectly, for a specific course—say, CS-347. That is, we wish to find a course

[6]This instance of *prereq* differs from that used earlier for reasons that will become apparent as we use it to explain recursive queries.

Telegram: @uni_k

www.konkur.in

that is a direct prerequisite for CS-347, or is a prerequisite for a course that is a prerequisite for CS-347, and so on.

Thus, since CS-319 is a prerequisite for CS-347 and CS-315 and CS-101 are prerequisites for CS-319, CS-315 and CS-101 are also prerequisites (indirectly) for CS-347. Then, since CS-190 is a prerequisite for CS-315, CS-190 is another indirect prerequisite for CS-347. Continuing, we see that CS-101 is a prerequisite for CS-190, but note that CS-101 was already added to the list of prerequisites for CS-347. In a real university, rather than our example, we would not expect such a complex prerequisite structure, but this example serves to show some of the situations that might possibly arise.

The **transitive closure** of the relation *prereq* is a relation that contains all pairs (*cid*, *pre*) such that *pre* is a direct or indirect prerequisite of *cid*. There are numerous applications that require computation of similar transitive closures on **hierarchies**. For instance, organizations typically consist of several levels of organizational units. Machines consist of parts that in turn have subparts, and so on; for example, a bicycle may have subparts such as wheels and pedals, which in turn have subparts such as tires, rims, and spokes. Transitive closure can be used on such hierarchies to find, for example, all parts in a bicycle.

5.4.1 Transitive Closure Using Iteration

One way to write the preceding query is to use iteration: First find those courses that are a direct prerequisite of CS-347, then those courses that are a prerequisite of all the courses under the first set, and so on. This iterative process continues until we reach an iteration where no courses are added. Figure 5.14 shows a function *findAllPrereqs*(*cid*) to carry out this task; the function takes the *course_id* of the course as a parameter (*cid*), computes the set of all direct and indirect prerequisites of that course, and returns the set.

The procedure uses three temporary tables:

- *c_prereq*: stores the set of tuples to be returned.
- *new_c_prereq*: stores the courses found in the previous iteration.
- *temp*: used as temporary storage while sets of courses are manipulated.

Note that SQL allows the creation of temporary tables using the command **create temporary table**; such tables are available only within the transaction executing the query and are dropped when the transaction finishes. Moreover, if two instances of *findAllPrereqs* run concurrently, each gets its own copy of the temporary tables; if they shared a copy, their result could be incorrect.

The procedure inserts all direct prerequisites of course *cid* into *new_c_prereq* before the **repeat** loop. The **repeat** loop first adds all courses in *new_c_prereq* to *c_prereq*. Next, it computes prerequisites of all those courses in *new_c_prereq*, except those that have already been found to be prerequisites of *cid*, and stores them in the temporary table

Telegram: @uni_k

www.konkur.in


```
create function findAllPrereqs(cid varchar(8))
    - - Finds all courses that are prerequisite (directly or indirectly) for cid
returns table (course_id varchar(8))
    - - The relation prereq(course_id, prereq_id) specifies which course is
    - - directly a prerequisite for another course.
begin
    create temporary table c_prereq (course_id varchar(8));
        - - table c_prereq stores the set of courses to be returned
    create temporary table new_c_prereq (course_id varchar(8));
        - - table new_c_prereq contains courses found in the previous iteration
    create temporary table temp (course_id varchar(8));
        - - table temp is used to store intermediate results
    insert into new_c_prereq
        select prereq_id
        from prereq
        where course_id = cid;
    repeat
        insert into c_prereq
            select course_id
            from new_c_prereq;

        insert into temp
            (select prereq.prereq_id
                from new_c_prereq, prereq
                where new_c_prereq.course_id = prereq.course_id
            )
            except (
                select course_id
                from c_prereq
            );
        delete from new_c_prereq;
        insert into new_c_prereq
            select *
            from temp;
        delete from temp;

    until not exists (select * from new_c_prereq)
    end repeat;
    return table c_prereq;
end
```

Figure 5.14 Finding all prerequisites of a course.

Telegram: @uni_k

www.konkur.in

0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

Figure 5.15 Prerequisites of CS-347 in iterations of function *findAllPrereqs*.

temp. Finally, it replaces the contents of *new_c_prereq* with the contents of *temp*. The **repeat** loop terminates when it finds no new (indirect) prerequisites.

Figure 5.15 shows the prerequisites that are found in each iteration when the procedure is called for CS-347. While *c_prereq* could have been updated in one SQL statement, we need first to construct *new_c_prereq* so we can tell when nothing is being added in the (final) iteration.

The use of the **except** clause in the function ensures that the function works even in the (abnormal) case where there is a cycle of prerequisites. For example, if *a* is a prerequisite for *b*, *b* is a prerequisite for *c*, and *c* is a prerequisite for *a*, there is a cycle.

While cycles may be unrealistic in course prerequisites, cycles are possible in other applications. For instance, suppose we have a relation *flights*(*to, from*) that says which cities can be reached from which other cities by a direct flight. We can write code similar to that in the *findAllPrereqs* function, to find all cities that are reachable by a sequence of one or more flights from a given city. All we have to do is to replace *prereq* with *flight* and replace attribute names correspondingly. In this situation, there can be cycles of reachability, but the function would work correctly since it would eliminate cities that have already been seen.

5.4.2 Recursion in SQL

It is rather inconvenient to specify transitive closure using iteration. There is an alternative approach, using recursive view definitions, that is easier to use.

We can use recursion to define the set of courses that are prerequisites of a particular course, say CS-347, as follows. The courses that are prerequisites (directly or indirectly) of CS-347 are:

- Courses that are prerequisites for CS-347.
- Courses that are prerequisites for those courses that are prerequisites (directly or indirectly) for CS-347.

Note that case 2 is recursive, since it defines the set of courses that are prerequisites of CS-347 in terms of the set of courses that are prerequisites of CS-347. Other examples

Telegram: @uni_k

www.konkur.in


```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id
        from rec_prereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

Figure 5.16 Recursive query in SQL.

of transitive closure, such as finding all subparts (direct or indirect) of a given part can also be defined in a similar manner, recursively.

The SQL standard supports a limited form of recursion, using the **with recursive** clause, where a view (or temporary view) is expressed in terms of itself. Recursive queries can be used, for example, to express transitive closure concisely. Recall that the **with** clause is used to define a temporary view whose definition is available only to the query in which it is defined. The additional keyword **recursive** specifies that the view is recursive.[7]

For example, we can find every pair (*cid*,*pre*) such that *pre* is directly or indirectly a prerequisite for course *cid*, using the recursive SQL view shown in Figure 5.16.

Any recursive view must be defined as the union[8] of two subqueries: a **base query** that is nonrecursive and a **recursive query** that uses the recursive view. In the example in Figure 5.16, the base query is the select on *prereq* while the recursive query computes the join of *prereq* and *rec_prereq*.

The meaning of a recursive view is best understood as follows: First compute the base query and add all the resultant tuples to the recursively defined view relation *rec_prereq* (which is initially empty). Next compute the recursive query using the current contents of the view relation, and add all the resulting tuples back to the view relation. Keep repeating the above step until no new tuples are added to the view relation. The resultant view relation instance is called a **fixed point** of the recursive view definition. (The term "fixed" refers to the fact that there is no further change.) The view relation is thus defined to contain exactly the tuples in the fixed-point instance.

Applying this logic to our example, we first find all direct prerequisites of each course by executing the base query. The recursive query adds one more level of courses

[7] Some systems treat the **recursive** keyword as optional; others disallow it.
[8] Some systems, notably Oracle, require use of **union all**.

Telegram: @uni_k

www.konkur.in

in each iteration, until the maximum depth of the course-prereq relationship is reached. At this point no new tuples are added to the view, and a fixed point is reached.

To find the prerequisites of a specific course, such as CS-347, we can modify the outer level query by adding a **where** clause "**where** *rec_prereq.course_id* = 'CS-347'". One way to evaluate the query with the selection is to compute the full contents of *rec_prereq* using the iterative technique, and then select from this result only those tuples whose *course_id* is CS-347. However, this would result in computing (course, prerequisite) pairs for all courses, all of which are irrelevant except for those for the course CS-347. In fact the database system is not required to use this iterative technique to compute the full result of the recursive query and then perform the selection. It may get the same result using other techniques that may be more efficient, such as that used in the function *findAllPrereqs* which we saw earlier. See the bibliographic notes for references to more information on this topic.

There are some restrictions on the recursive query in a recursive view; specifically, the query must be **monotonic**, that is, its result on a view relation instance V_1 must be a superset of its result on a view relation instance V_2 if V_1 is a superset of V_2. Intuitively, if more tuples are added to the view relation, the recursive query must return at least the same set of tuples as before, and possibly return additional tuples.

In particular, recursive queries may not use any of the following constructs, since they would make the query nonmonotonic:

- Aggregation on the recursive view.
- **not exists** on a subquery that uses the recursive view.
- Set difference (**except**) whose right-hand side uses the recursive view.

For instance, if the recursive query was of the form $r - v$, where v is the recursive view, if we add a tuple to v, the result of the query can become smaller; the query is therefore not monotonic.

The meaning of recursive views can be defined by the iterative procedure as long as the recursive query is monotonic; if the recursive query is nonmonotonic, the meaning of the view is hard to define. SQL therefore requires the queries to be monotonic. Recursive queries are discussed in more detail in the context of the Datalog query language, in Section 27.4.6.

SQL also allows creation of recursively defined permanent views by using **create recursive view** in place of **with recursive**. Some implementations support recursive queries using a different syntax. This includes the Oracle **start with** / **connect by prior** syntax for what it calls hierarchical queries. [9] See the respective system manuals for further details.

[9] Staring with Oracle 12.c, the standard syntax is accepted in addition to the legacy hierarchical syntax, with the **recursive** keyword omitted and with the requirement in our example that **union all** be

Telegram: @uni_k

www.konkur.in

5.5 Advanced Aggregation Features

The aggregation support in SQL is quite powerful and handles most common tasks with ease. However, there are some tasks that are hard to implement efficiently with the basic aggregation features. In this section, we study features in SQL to handle some such tasks.

5.5.1 Ranking

Finding the position of a value within a set is a common operation. For instance, we may wish to assign students a rank in class based on their grade-point average (GPA), with the rank 1 going to the student with the highest GPA, the rank 2 to the student with the next highest GPA, and so on. A related type of query is to find the percentile in which a value in a (multi)set belongs, for example, the bottom third, middle third, or top third. While such queries can be expressed using the SQL constructs we have seen so far, they are difficult to express and inefficient to evaluate. Programmers may resort to writing the query partly in SQL and partly in a programming language. We study SQL support for direct expression of these types of queries here.

In our university example, the *takes* relation shows the grade each student earned in each course taken. To illustrate ranking, let us assume we have a view *student_grades* (*ID, GPA*) giving the grade-point average of each student.[10]

Ranking is done with an **order by** specification. The following query gives the rank of each student:

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades;
```

Note that the order of tuples in the output is not defined, so they may not be sorted by rank. An extra **order by** clause is needed to get them in sorted order, as follows:

```
select ID, rank () over (order by (GPA) desc) as s_rank
from student_grades
order by s_rank;
```

A basic issue with ranking is how to deal with the case of multiple tuples that are the same on the ordering attribute(s). In our example, this means deciding what to do if there are two students with the same GPA. The **rank** function gives the same rank to all tuples that are equal on the **order by** attributes. For instance, if the highest GPA is shared by two students, both would get rank 1. The next rank given would be 3, not 2, so if three students get the next highest GPA, they would all get rank 3, and the next

[10] The SQL statement to create the view *student_grades* is somewhat complex since we must convert the letter grades in the *takes* relation to numbers and weight the grades for each course by the number of credits for that course. The definition of this view is the goal of Exercise 4.6.

Telegram: @uni_k

www.konkur.in

student(s) would get rank 6, and so on. There is also a **dense_rank** function that does not create gaps in the ordering. In the preceding example, the tuples with the second highest value all get rank 2, and tuples with the third highest value get rank 3, and so on.

If there are null values among the values being ranked, they are treated as the highest values. That makes sense in some situations, although for our example, it would result in students with no courses being shown as having the highest GPAs. Thus, we see that care needs to be taken in writing ranking queries in cases where null values may appear. SQL permits the user to specify where they should occur by using **nulls first** or **nulls last**, for instance:

```
select ID, rank () over (order by GPA desc nulls last) as s_rank
from student_grades;
```

It is possible to express the preceding query with the basic SQL aggregation functions, using the following query:

```
select ID, (1 + (select count(*)
                 from student_grades B
                 where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

It should be clear that the rank of a student is merely 1 plus the number of students with a higher *GPA*, which is exactly what the query specifies.[11] However, this computation of each student's rank takes time linear in the size of the relation, leading to an overall time quadratic in the size of the relation. On large relations, the above query could take a very long time to execute. In contrast, the system's implementation of the **rank** clause can sort the relation and compute the rank in much less time.

Ranking can be done within partitions of the data. For instance, suppose we wish to rank students by department rather than across the entire university. Assume that a view is defined like *student_grades* but including the department name: *dept_grades*(*ID, dept_name, GPA*). The following query then gives the rank of students within each section:

```
select ID, dept_name,
    rank () over (partition by dept_name order by GPA desc) as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

[11] There is a slight technical difference if a student has not taken any courses and therefore has a *null* GPA. Due to how comparisons of null values work in SQL, a student with a null GPA does not

Telegram: @uni_k

www.konkur.in

The outer **order by** clause orders the result tuples by department name, and within each department by the rank.

Multiple **rank** expressions can be used within a single **select** statement; thus, we can obtain the overall rank and the rank within the department by using two **rank** expressions in the same **select** clause. When ranking (possibly with partitioning) occurs along with a **group by** clause, the **group by** clause is applied first, and partitioning and ranking are done on the results of the **group by**. Thus, aggregate values can then be used for ranking.

It is often the case, especially for large results, that we may be interested only in the top-ranking tuples of the result rather than the entire list. For rank queries, this can be done by nesting the ranking query within a containing query whose **where** clause chooses only those tuples whose rank is lower than some specified value. For example, to find the top 5 ranking students based on GPA we could extend our earlier example by writing:

```
select *
from (select ID, rank() over (order by (GPA) desc) as s_rank
      from student_grades)
where s_rank <= 5;
```

This query does not necessarily give 5 students, since there could be ties. For example, if 2 students tie for fifth, the result would contain a total of 6 tuples. Note that the bottom n is simply the same as the top n with a reverse sorting order.

Several database systems provide nonstandard SQL syntax to specify directly that only the top n results are required. In our example, this would allow us to find the top 5 students without the need to use the **rank** function. However, those constructs result in exactly the number of tuples specified (5 in our example), and so ties for the final position are broken arbitrarily. The exact syntax for these "top n" queries varies widely among systems; see Note 5.4 on page 222. Note that the top n constructs do not support partitioning; so we cannot get the top n within each partition without performing ranking.

Several other functions can be used in place of **rank**. For instance, **percent_rank** of a tuple gives the rank of the tuple as a fraction. If there are n tuples in the partition[12] and the rank of the tuple is r, then its percent rank is defined as $(r - 1)/(n - 1)$ (and as *null* if there is only one tuple in the partition). The function **cume_dist**, short for cumulative distribution, for a tuple is defined as p/n where p is the number of tuples in the partition with ordering values preceding or equal to the ordering value of the tuple and n is the number of tuples in the partition. The function **row_number** sorts the rows and gives each row a unique number corresponding to its position in the sort order; different rows with the same ordering value would get different row numbers, in a nondeterministic fashion.

[12]The entire set is treated as a single partition if no explicit

Telegram: @uni_k

www.konkur.in



Telegram: @uni_k

www.konkur.in

Finally, for a given constant n, the ranking function **ntile**(n) takes the tuples in each partition in the specified order and divides them into n buckets with equal numbers of tuples.[13] For each tuple, **ntile**(n) then gives the number of the bucket in which it is placed, with bucket numbers starting with 1. This function is particularly useful for constructing histograms based on percentiles. We can show the quartile into which each student falls based on GPA by the following query:

```
select ID, ntile(4) over (order by (GPA desc)) as quartile
from student_grades;
```

5.5.2 Windowing

Window queries compute an aggregate function over ranges of tuples. This is useful, for example, to compute an aggregate of a fixed range of time; the time range is called a *window*. Windows may overlap, in which case a tuple may contribute to more than one window. This is unlike the partitions we saw earlier, where a tuple could contribute to only one partition.

An example of the use of windowing is trend analysis. Consider our earlier sales example. Sales may fluctuate widely from day to day based on factors like weather (e.g., a snowstorm, flood, hurricane, or earthquake might reduce sales for a period of time). However, over a sufficiently long period of time, fluctuations might be less (continuing the example, sales may "make up" for weather-related downturns). Stock-market trend analysis is another example of the use of the windowing concept. Various "moving averages" are found on business and investment web sites.

It is relatively easy to write an SQL query using those features we have already studied to compute an aggregate over one window, for example, sales over a fixed 3-day period. However, if we want to do this for *every* 3-day period, the query becomes cumbersome.

SQL provides a windowing feature to support such queries. Suppose we are given a view *tot_credits* (*year, num_credits*) giving the total number of credits taken by students in each year.[14] Note that this relation can contain at most one tuple for each year. Consider the following query:

```
select year, avg(num_credits)
                over (order by year rows 3 preceding)
                as avg_total_credits
from tot_credits;
```

[13] If the total number of tuples in a partition is not divisible by n, then the number of tuples in each bucket can differ by at most 1. Tuples with the same value for the ordering attribute may be assigned to different buckets, nondeterministically, in order to make the number of tuples in each bucket equal.

[14] We leave the definition of this view in terms of our university

Telegram: @uni_k

www.konkur.in

This query computes averages over the three *preceding* tuples in the specified sort order. Thus, for 2019, if tuples for years 2018 and 2017 are present in the relation *tot_credits*, since each year is represented by only one tuple, the result of the window definition is the average of the values for years 2017, 2018, and 2019. The averages each year would be computed in a similar manner. For the earliest year in the relation *tot_credits*, the average would be over only that year itself, while for the next year, the average would be over 2 years. Note that this example makes sense only because each year appears only once in *tot_weight*. Were this not the case, then there would be several possible orderings of tuples since tuples for the same year could be in any order. We shall see shortly a windowing query that uses a range of values instead of a specific number of tuples.

Suppose that instead of going back a fixed number of tuples, we want the window to consist of all prior years. That means the number of prior years considered is not fixed. To get the average total credits over all prior years, we write:

```
select year, avg(num_credits)
                over (order by year rows unbounded preceding)
                as avg_total_credits
from tot_credits;
```

It is possible to use the keyword **following** in place of **preceding**. If we did this in our example, the *year* value specifies the beginning of the window instead of the end. Similarly, we can specify a window beginning before the current tuple and ending after it:

```
select year, avg(num_credits)
            over (order by year rows between 3 preceding and 2 following)
            as avg_total_credits
from tot_credits;
```

In our example, all tuples pertain to the entire university. Suppose instead we have credit data for each department in a view *tot_credits_dept* (*dept_name, year, num_credits*) giving the total number of credits students took with the particular department in the specified year. (Again, we leave writing this view definition as an exercise.) We can write windowing queries that treat each department separately by partitioning by *dept_name*:

```
select dept_name, year, avg(num_credits)
            over (partition by dept_name
                order by year rows between 3 preceding and current row)
            as avg_total_credits
from tot_credits_dept;
```

Telegram: @uni_k

www.konkur.in

dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3

Figure 5.17 An example of *sales* relation.

Telegram: @uni_k

www.konkur.in

dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3

Figure 5.18 Result of SQL pivot operation on the *sales* relation of Figure 5.17.

The use of the keyword **range** in place of **row** allows the windowing query to cover all tuples with a particular value rather than covering a specific number of tuples. Thus for example, **rows current row** refers to exactly one tuple, while **range current row** refers to all tuples whose value for the *sort* attribute is the same as that of the current tuple. The **range** keyword is not implemented fully in every system.[15]

5.5.3 Pivoting

Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their item_name, color, and size, and that we have a relation *sales* with the schema.

sales (*item_name*, *color*, *clothes_size*, *quantity*)

Suppose that *item_name* can take on the values (skirt, dress, shirt, pants), *color* can take on the values (dark, pastel, white), *clothes_size* can take on values (small, medium, large), and *quantity* is an integer value representing the total number of items sold of a given (*item_name*, *color*, *clothes_size*) combination. An instance of the *sales* relation is shown in Figure 5.17.

Figure 5.18 shows an alternative way to view the data that is present in Figure 5.17; the values "dark", "pastel", and "white" of attribute *color* have become attribute names in Figure 5.18. The table in Figure 5.18 is an example of a **cross-tabulation** (or **cross-tab**, for short), also referred to as a **pivot-table**.

The values of the new attributes *dark, pastel* and *white* in our example are defined as follows. For a particular combination of *item_name*, *clothes_size* (e.g., ("dress", "dark"))

[15] Some systems, such as PostgreSQL, allow **range** only with **unbounded**.

Telegram: @uni_k

www.konkur.in

if there is a single tuple with *color* value "dark", the *quantity* value of that attribute appears as the value for the attribute *dark*. If there are multiple such tuples, the values are aggregated using the **sum** aggregate in our example; in general other aggregate functions could be used instead. Values for the other two attributes, *pastel* and *white*, are similarly defined.

In general, a cross-tab is a table derived from a relation (say, *R*), where values for some attribute of relation *R* (say, *A*) become attribute names in the result; the attribute *A* is the **pivot** attribute. Cross-tabs are widely used for data analysis, and are discussed in more detail in Section 11.3.

Several SQL implementations, such as Microsoft SQL Server, and Oracle, support a **pivot** clause that allows creation of cross-tabs. Given the *sales* relation from Figure 5.17, the query:

```
select *
from sales
pivot (
        sum(quantity)
        for color in ('dark', 'pastel', 'white')
)
```

returns the result shown in Figure 5.18.

Note that the **for** clause within the **pivot** clause specifies (i) a pivot attribute (*color*, in the above query), (ii) the values of that attribute that should appear as attribute names in the pivot result (dark, pastel and white, in the above query), and (iii) the aggregate function that should be used to compute the value of the new attributes (aggregate function **sum**, on the attribute *quantity*, in the above query).

The attribute *color* and *quantity* do not appear in the result, but all other attributes are retained. In case more than one tuple contributes values to a given cell, the aggregate operation within the **pivot** clause specifies how the values should be combined. In the above example, the *quantity* values are aggregated using the **sum** function.

A query using **pivot** can be written using basic SQL constructs, without using the pivot construct, but the construct simplifies the task of writing such queries.

5.5.4 Rollup and Cube

SQL supports generalizations of the **group by** construct using the **rollup** and **cube** operations, which allow multiple **group by** queries to be run in a single query, with the result returned as a single relation.

Consider again our retail shop example and the relation:

sales (*item_name*, *color*, *clothes_size*, *quantity*)

We can find the number of items sold in each item name by writing a simple **group by** query:

Telegram: @uni_k

www.konkur.in


```
select item_name, sum(quantity) as quantity
from sales
group by item_name;
```

Similarly, we can find the number of items sold in each color, and each size. We can further find a breakdown of sales by item-name and color by writing:

```
select item_name, color, sum(quantity) as quantity
from sales
group by item_name, color ;
```

Similarly, a query with **group by** *item_name*, *color*, *clothes_size* would allow us to see the sales breakdown by (*item_name*, *color*, *clothes_size*) combinations.

Data analysts often need to view data aggregated in multiple ways as illustrated above. The SQL **rollup** and **cube** constructs provide a concise way to get multiple such aggregates using a single query, instead of writing multiple queries.

The **rollup** construct is illustrated using the following query:

```
select item_name, color, sum(quantity)
from sales
group by rollup(item_name, color);
```

The result of the query is shown in Figure 5.19. The above query is equivalent to the following query using the **union** operation.

```
(select item_name, color, sum(quantity) as quantity
from sales
group by item_name, color)
union
(select item_name, null as color, sum(quantity) as quantity
from sales
group by item_name)
union
(select null as item_name, null as color, sum(quantity) as quantity
from sales)
```

The construct **group by rollup**(*item_name*, *color*) generates 3 groupings:

{ (*item_name*, *color*), (*item_name*), () }

where () denotes an empty **group by** list. Observe that a grouping is present for each prefix of the attributes listed in the **rollup** clause, including the empty prefix. The query result contains the union of the results by these groupings. The different groupings generate different schemas; to bring the results of the different groupings to a common

Telegram: @uni_k

www.konkur.in

skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5
skirt	*null*	53
dress	*null*	35
shirt	*null*	49
pants	*null*	27
null	*null*	164

Figure 5.19 Query result: **group by rollup** (*item_name, color*).

schema, tuples in the result contain *null* as the value of those attributes not present in a particular grouping.[16]

The **cube** construct generates an even larger number of groupings, consisting of *all subsets* of the attributes listed in the **cube** construct. For example, the query:

select *item_name*, *color*, *clothes_size*, **sum**(*quantity*)
from *sales*
group by cube(*item_name*, *color*, *clothes_size*);

generates the following groupings:

{ (*item_name*, *color*, *clothes_size*), (*item_name*, *color*), (*item_name*, *clothes_size*), (*color*, *clothes_size*), (*item_name*), (*color*), (*clothes_size*), ( ) }

To bring the results of the different groupings to a common schema, as with **rollup**, tuples in the result contain *null* as the value of those attributes not present in a particular grouping.

[16]The SQL **outer union** operation can be used to perform a union of relations that may not have a common schema. The resultant schema has the union of all the attributes across the inputs; each input tuple is mapped to an output tuple by adding all the attributes missing in that tuple, with the value set to null. Our union query can be written using outer union, and in that case we do not need to explicitly generate null-value attributes using *null* **as** attribute-name constructs, as we have done in the above query.

Telegram: @uni_k

www.konkur.in

Multiple **rollup**s and **cube**s can be used in a single **group by** clause. For instance, the following query:

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name), rollup(color, clothes_size);
```

generates the groupings:

{ (*item_name*, *color*, *clothes_size*), (*item_name*, *color*), (*item_name*), (*color*, *clothes_size*), (*color*), () }

To understand why, observe that **rollup**(*item_name*) generates a set of two groupings, {(*item_name*), ()}, while **rollup**(*color*, *clothes_size*) generates a set of three groupings, {(*color*, *clothes_size*), (*color*), () }. The Cartesian product of the two sets gives us the six groupings shown.

Neither the **rollup** nor the **cube** clause gives complete control on the groupings that are generated. For instance, we cannot use them to specify that we want only groupings {(*color*, *clothes_size*), (*clothes_size*, *item_name*)}. Such restricted groupings can be generated by using the **grouping sets** construct, in which one can specify the specific list of groupings to be used. To obtain only groupings {(*color*, *clothes_size*), (*clothes_size*, *item_name*)}, we would write:

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by grouping sets ((color, clothes_size), (clothes_size, item_name));
```

Analysts may want to distinguish those nulls generated by **rollup** and **cube** operations from "normal" nulls actually stored in the database or arising from an outer join. The **grouping**() function returns 1 if its argument is a null value generated by a **rollup** or **cube** and 0 otherwise (note that the **grouping** function is different from the **grouping sets** construct). If we wish to display the **rollup** query result shown in Figure 5.19, but using the value "all" in place of nulls generated by **rollup**, we can use the query:

```
select (case when grouping(item_name) = 1 then 'all'
             else item_name end) as item_name,
       (case when grouping(color) = 1 then 'all'
             else color end) as color,
sum(quantity) as quantity
from sales
group by rollup(item_name, color);
```

One might consider using the following query using **coalesce**, but it would incorrectly convert null item names and colors to **all**:

Telegram: @uni_k

www.konkur.in


```
select coalesce (item_name,'all') as item_name,
        coalesce (color,'all') as color,
sum(quantity) as quantity
from sales
group by rollup(item_name, color);
```

5.6 Summary

- SQL queries can be invoked from host languages via embedded and dynamic SQL. The ODBC and JDBC standards define application program interfaces to access SQL databases from C and Java language programs.
- Functions and procedures can be defined using SQL procedural extensions that allow iteration and conditional (if-then-else) statements.
- Triggers define actions to be executed automatically when certain events occur and corresponding conditions are satisfied. Triggers have many uses, such as business rule implementation and audit logging. They may carry out actions outside the database system by means of external language routines.
- Some queries, such as transitive closure, can be expressed either by using iteration or by using recursive SQL queries. Recursion can be expressed using either recursive views or recursive **with** clause definitions.
- SQL supports several advanced aggregation features, including ranking and windowing queries, as well as pivot, and rollup/cube operations. These simplify the expression of some aggregates and allow more efficient evaluation.

Review Terms

- JDBC
- Prepared statements
- SQL injection
- Metadata
- Updatable result sets
- Open Database Connectivity (ODBC)
- Embedded SQL
- Embedded database
- Stored procedures and functions
- Table functions.
- Parameterized views
- Persistent Storage Module (PSM).
- Exception conditions
- Handlers
- External language routines
- Sandbox
- Trigger
- Transitive closure
- Hierarchies

Telegram: @uni_k

www.konkur.in

- Create temporary table
- Base query
- Recursive query
- Fixed point
- Monotonic
- Windowing
- Ranking functions
- Cross-tabulation
- Cross-tab
- Pivot-table
- Pivot
- SQL **group by cube**, **group by rollup**

Practice Exercises

5.1 Consider the following relations for a company database:

- *emp* (*ename*, *dname*, *salary*)
- *mgr* (*ename*, *mname*)

and the Java code in Figure 5.20, which uses the JDBC API. Assume that the userid, password, machine name, etc. are all okay. Describe in concise English what the Java program does. (That is, produce an English sentence like "It finds the manager of the toy department," not a line-by-line description of what each Java statement does.)

5.2 Write a Java method using JDBC metadata features that takes a `ResultSet` as an input parameter and prints out the result in tabular form, with appropriate names as column headings.

5.3 Suppose that we wish to find all courses that must be taken before some given course. That means finding not only the prerequisites of that course, but prerequisites of prerequisites, and so on. Write a complete Java program using JDBC that:

- Takes a *course_id* value from the keyboard.
- Finds prerequisites of that course using an SQL query submitted via JDBC.
- For each course returned, finds its prerequisites and continues this process iteratively until no new prerequisite courses are found.
- Prints out the result.

For this exercise, do not use a recursive SQL query, but rather use the iterative approach described previously. A well-developed solution will be robust to the error case where a university has accidentally created a cycle of prerequisites (that is, for example, course *A* is a prerequisite for course *B*, course *B* is a prerequisite for course *C*, and course *C* is a prerequisite for course *A*).

Telegram: @uni_k

www.konkur.in


```
import java.sql.*;
public class Mystery {
   public static void main(String[] args) {
       try (
           Connection con=DriverManager.getConnection(
               "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
           q = "select mname from mgr where ename = ?";
           PreparedStatement stmt=con.prepareStatement();
       )
       {
           String q;
           String empName = "dog";
           boolean more;
           ResultSet result;
           do {
               stmt.setString(1, empName);
               result = stmt.executeQuery(q);
               more = result.next();
               if (more) {
                   empName = result.getString("mname");
                   System.out.println (empName);
               }
           } while (more);
           s.close();
           con.close();
       }
       catch(Exception e){
           e.printStackTrace();
       }
   }
}
```

Figure 5.20 Java code for Exercise 5.1 (using Oracle JDBC).

5.4 Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.

5.5 Show how to enforce the constraint "an instructor cannot teach two different sections in a semester in the same time slot." using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).

Telegram: @uni_k

www.konkur.in

branch (*branch_name*, *branch_city, assets*)
customer (*customer_name*, *customer_street, cust omer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name, balance*)
depositor (*customer_name*, *account_number*)

Figure 5.21 Banking database for Exercise 5.6.

5.6 Consider the bank database of Figure 5.21. Let us define a view *branch_cust* as follows:

```
create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account_number = account.account_number
```

Suppose that the view is *materialized*; that is, the view is computed and stored. Write triggers to *maintain* the view, that is, to keep it up-to-date on insertions to *depositor* or *account*. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

5.7 Consider the bank database of Figure 5.21. Write an SQL trigger to carry out the following action: On **delete** of an account, for each customer-owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.

5.8 Given a relation $S(student, subject, marks)$, write a query to find the top 10 students by total marks, by using SQL ranking. Include all students tied for the final spot in the ranking, even if that results in more than 10 total students.

5.9 Given a relation *nyse*(*year, month, day, shares_traded, dollar_volume*) with trading data from the New York Stock Exchange, list each trading day in order of number of shares traded, and show each day's rank.

5.10 Using the relation from Exercise 5.9, write an SQL query to generate a report showing the number of shares traded, number of trades, and total dollar volume broken down by year, each month of each year, and each trading day.

5.11 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.

Telegram: @uni_k

www.konkur.in

Exercises

5.12 Write a Java program that allows university administrators to print the teaching record of an instructor.

a. Start by having the user input the login *ID* and password; then open the proper connection.

b. The user is asked next for a search substring and the system returns (ID, *name*) pairs of instructors whose names match the substring. Use the **like** ('%substring%') construct in SQL to do this. If the search comes back empty, allow continued searches until there is a nonempty result.

c. Then the user is asked to enter an ID number, which is a number between 0 and 99999. Once a valid number is entered, check if an instructor with that ID exists. If there is no instructor with the given ID, print a reasonable message and quit.

d. If the instructor has taught no courses, print a message saying that. Otherwise print the teaching record for the instructor, showing the department name, course identifier, course title, section number, semester, year, and total enrollment (and sort those by *dept_name, course_id, year, semester*).

Test carefully for bad input. Make sure your SQL queries won't throw an exception. At login, exceptions may occur since the user might type a bad password, but catch those exceptions and allow the user to try again.

5.13 Suppose you were asked to define a class `MetaDisplay` in Java, containing a method `static void printTable(String r)`; the method takes a relation name *r* as input, executes the query "**select** * **from** *r*", and prints the result out in tabular format, with the attribute names displayed in the header of the table.

a. What do you need to know about relation *r* to be able to print the result in the specified tabular format?

b. What JDBC methods(s) can get you the required information?

c. Write the method `printTable(String r)` using the JDBC API.

5.14 Repeat Exercise 5.13 using ODBC, defining `void printTable(char *r)` as a function instead of a method.

5.15 Consider an employee database with two relations

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company name*, *salary*)

Telegram: @uni_k

www.konkur.in

where the primary keys are underlined. Write a function *avg_salary* that takes a company name as an argument and finds the average salary of employees at that company. Then, write an SQL statement, using that function, to find companies whose employees earn a higher salary, on average, than the average salary at "First Bank".

5.16 Consider the relational schema

part(*part_id*, *name*, *cost*)
subpart(*part_id*, *subpart_id*, *count*)

where the primary-key attributes are underlined. A tuple $(p_1, p_2, 3)$ in the *subpart* relation denotes that the part with *part_id* p_2 is a direct subpart of the part with *part_id* p_1, and p_1 has 3 copies of p_2. Note that p_2 may itself have further subparts. Write a recursive SQL query that outputs the names of all subparts of the part with part-id 'P-100'.

5.17 Consider the relational schema from Exercise 5.16. Write a JDBC function using nonrecursive SQL to find the total cost of part "P-100", including the costs of all its subparts. Be sure to take into account the fact that a part may have multiple occurrences of a subpart. You may use recursion in Java if you wish.

5.18 Redo Exercise 5.12 using the language of your database system for coding stored procedures and functions. Note that you are likely to have to consult the online documentation for your system as a reference, since most systems use syntax differing from the SQL standard version followed in the text. Specifically, write a prodedure that takes an instructor *ID* as an argument and produces printed output in the format specified in Exercise 5.12, or an appropriate message if the instructor does not exist or has taught no courses. (For a simpler version of this exercise, rather than providing printed output, assume a relation with the appropriate schema and insert your answer there without worrying about testing for erroneous argument values.)

5.19 Suppose there are two relations *r* and *s*, such that the foreign key *B* of *r* references the primary key *A* of *s*. Describe how the trigger mechanism can be used to implement the **on delete cascade** option when a tuple is deleted from *s*.

5.20 The execution of a trigger can cause another action to be triggered. Most database systems place a limit on how deep the nesting can be. Explain why they might place such a limit.

5.21 Modify the recursive query in Figure 5.16 to define a relation

prereq_depth(*course_id*, *prereq id*, *depth*)

Telegram: @uni_k

www.konkur.in

Garfield	359	A	BIO-101	1
Garfield	359	B	BIO-101	2
Saucon	651	A	CS-101	2
Saucon	550	C	CS-319	1
Painter	705	D	MU-199	1
Painter	403	D	FIN-201	1

Figure 5.22 The relation *r* for Exercise 5.24.

where the attribute *depth* indicates how many levels of intermediate prerequisites there are between the course and the prerequisite. Direct prerequisites have a depth of 0. Note that a prerequisite course may have multiple depths and thus may appear more than once.

5.22 Given relation $s(a, b, c)$, write an SQL statement to generate a histogram showing the sum of c values versus a, dividing a into 20 equal-sized partitions (i.e., where each partition contains 5 percent of the tuples in s, sorted by a).

5.23 Consider the *nyse* relation of Exercise 5.9. For each month of each year, show the total monthly dollar volume and the average monthly dollar volume for that month and the two prior months. (*Hint*: First write a query to find the total dollar volume for each month of each year. Once that is right, put that in the from clause of the outer query that solves the full problem. That outer query will need windowing. The subquery does not.)

5.24 Consider the relation, *r*, shown in Figure 5.22. Give the result of the following query:

```
select building, room_number, time_slot_id, count(*)
from r
group by rollup (building, room_number, time_slot_id)
```

Tools

We provide sample JDBC code on our book web site db-book.com.

Most database vendors, including IBM, Microsoft, and Oracle, provide OLAP tools as part of their database systems, or as add-on applications. Tools may be integrated with a larger "business intelligence" product such as IBM Cognos. Many companies also provide analysis tools for specific applications, such as customer relationship management (e.g., Oracle Siebel CRM).

Telegram: @uni_k

www.konkur.in

Further Reading

More details about JDBC may be found at docs.oracle.com/javase/tutorial/jdbc.

In order to write stored procedures, stored functions, and triggers that can be executed on a given system, you need to refer to the system documentation.

Although our discussion of recursive queries focused on SQL syntax, there are other approaches to recursion in relational databases. Datalog is a database language based on the Prolog programming language and is described in more detail in Section 27.4 (available online).

OLAP features in SQL, including rollup, and cubes were introduced in SQL:1999, and window functions with ranking and partitioning were added in SQL:2003. OLAP features, including window functions, are supported by most databases today. Although most follow the SQL standard syntax that we have presented, there are some differences; refer to the system manuals of the system that you are using for further details. Microsoft's Multidimensional Expressions (MDX) is an SQL-like query language designed for querying OLAP cubes.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in



PART 2

DATABASE DESIGN

The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data. The needs of the users play a central role in the design process. In this part, we focus primarily on the design of the database schema. We also outline some of the other design tasks.

The entity-relationship (E-R) model described in Chapter 6 is a high-level data model. Instead of representing all data in tables, it distinguishes between basic objects, called *entities*, and *relationships* among these objects. It is often used as a first step in database-schema design.

Relational database design—the design of the relational schema— was covered informally in earlier chapters. There are, however, principles that can be used to distinguish good database designs from bad ones. These are formalized by means of several "normal forms" that offer different trade-offs between the possibility of inconsistencies and the efficiency of certain queries. Chapter 7 describes the formal design of relational schemas.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



CHAPTER 6

Database Design Using the E-R Model

Up to this point in the text, we have assumed a given database schema and studied how queries and updates are expressed. We now consider how to design a database schema in the first place. In this chapter, we focus on the entity-relationship data model (E-R), which provides a means of identifying entities to be represented in the database and how those entities are related. Ultimately, the database design will be expressed in terms of a relational database design and an associated set of constraints. We show in this chapter how an E-R design can be transformed into a set of relation schemas and how some of the constraints can be captured in that design. Then, in Chapter 7, we consider in detail whether a set of relation schemas is a good or bad database design and study the process of creating good designs using a broader set of constraints. These two chapters cover the fundamental concepts of database design.

6.1 Overview of the Design Process

The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data. The needs of the users play a central role in the design process. In this chapter, we focus on the design of the database schema, although we briefly outline some of the other design tasks later in the chapter.

6.1.1 Design Phases

For small applications, it may be feasible for a database designer who understands the application requirements to decide directly on the relations to be created, their attributes, and constraints on the relations. However, such a direct design process is difficult for real-world applications, since they are often highly complex. Often no one person understands the complete data needs of an application. The database designer must interact with users of the application to understand the needs of the application, represent them in a high-level fashion that can be understood by the users, and

Telegram: @uni_k

www.konkur.in

then translate the requirements into lower levels of the design. A high-level data model serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, the data requirements of the database users, and a database structure that fulfills these requirements.

- The initial phase of database design is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements. While there are techniques for diagrammatically representing user requirements, in this chapter we restrict ourselves to textual descriptions of user requirements.
- Next, the designer chooses a data model and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise. The entity-relationship model, which we study in the rest of this chapter, is typically used to represent the conceptual design. Stated in terms of the entity-relationship model, the conceptual schema specifies the entities that are represented in the database, the attributes of the entities, the relationships among the entities, and constraints on the entities and relationships. Typically, the conceptual-design phase results in the creation of an entity-relationship diagram that provides a graphic representation of the schema.

  The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. She can also examine the design to remove any redundant features. Her focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.
- A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure that it meets functional requirements.
- The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.
  - In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The implementation data model is typically the relational data model, and this step typically consists of mapping the conceptual schema defined using the entity-relationship model into a relation schema.
  - Finally, the designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database

Telegram: @uni_k

www.konkur.in

are specified. These features include the form of file organization and choice of index structures, discussed in Chapter 13 and Chapter 14.

The physical schema of a database can be changed relatively easily after an application has been built. However, changes to the logical schema are usually harder to carry out, since they may affect a number of queries and updates scattered across application code. It is therefore important to carry out the database design phase with care, before building the rest of the database application.

6.1.2 Design Alternatives

A major part of the database design process is deciding how to represent in the design the various types of "things" such as people, places, products, and the like. We use the term *entity* to refer to any such distinctly identifiable item. In a university database, examples of entities would include instructors, students, departments, courses, and course offerings. We assume that a course may have run in multiple semesters, as well as multiple times in a semester; we refer to each such offering of a course as a section. The various entities are related to each other in a variety of ways, all of which need to be captured in the database design. For example, a student takes a course offering, while an instructor teaches a course offering; teaches and takes are examples of relationships between entities.

In designing a database schema, we must ensure that we avoid two major pitfalls:

1. **Redundancy:** A bad design may repeat information. For example, if we store the course identifier and title of a course with each course offering, the title would be stored redundantly (i.e., multiple times, unnecessarily) with each course offering. It would suffice to store only the course identifier with each course offering, and to associate the title with the course identifier only once, in a course entity.

   Redundancy can also occur in a relational schema. In the university example we have used so far, we have a relation with section information and a separate relation with course information. Suppose that instead we have a single relation where we repeat all of the course information (course_id, title, dept_name, credits) once for each section (offering) of the course. Information about courses would then be stored redundantly.

   The biggest problem with such redundant representation of information is that the copies of a piece of information can become inconsistent if the information is updated without taking precautions to update all copies of the information. For example, different offerings of a course may have the same course identifier, but may have different titles. It would then become unclear what the correct title of the course is. Ideally, information should appear in exactly one place.

2. **Incompleteness:** A bad design may make certain aspects of the enterprise difficult or impossible to model. For example, suppose that, as in case (1) above, we only had entities corresponding to course offering, without having an entity

Telegram: @uni_k

www.konkur.in

corresponding to courses. Equivalently, in terms of relations, suppose we have a single relation where we repeat all of the course information once for each section that the course is offered. It would then be impossible to represent information about a new course, unless that course is offered. We might try to make do with the problematic design by storing null values for the section information. Such a work-around is not only unattractive but may be prevented by primary-key constraints.

Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose. As a simple example, consider a customer who buys a product. Is the sale of this product a relationship between the customer and the product? Alternatively, is the sale itself an entity that is related both to the customer and to the product? This choice, though simple, may make an important difference in what aspects of the enterprise can be modeled well. Considering the need to make choices such as this for the large number of entities and relationships in a real-world enterprise, it is not hard to see that database design can be a challenging problem. Indeed we shall see that it requires a combination of both science and "good taste."

6.2 The Entity-Relationship Model

The **entity-relationship** (**E-R**) **data model** was developed to facilitate database design by allowing specification of an *enterprise schema* that represents the overall logical structure of a database.

The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model. The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes. The E-R model also has an associated diagrammatic representation, the E-R diagram. As we saw briefly in Section 1.3.1, an **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model.

The Tools section at the end of the chapter provides information about several diagram editors that you can use to create E-R diagrams.

6.2.1 Entity Sets

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties must uniquely identify an entity. For instance, a person may have a *person_id* property whose value uniquely identifies that person. Thus, the value 677-89-9011 for *person_id* would uniquely identify one particular person in the university. Similarly, courses can be thought of as entities, and *course_id* uniquely identifies a course entity in the university. An entity may be concrete, such

Telegram: @uni_k

www.konkur.in

as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set *instructor*. Similarly, the entity set *student* might represent the set of all students in the university.

In the process of modeling, we often use the term *entity set* in the abstract, without referring to a particular set of individual entities. We use the term **extension** of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms the extension of the entity set *instructor*. This distinction is similar to the difference between a relation and a relation instance, which we saw in Chapter 2.

Entity sets do not need to be disjoint. For example, it is possible to define the entity set *person* consisting of all people in a university. A *person* entity may be an *instructor* entity, a *student* entity, both, or neither.

An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *instructor* entity set are *ID*, *name*, *dept_name*, and *salary*. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we generally omit them to keep our examples simple. Possible attributes of the *course* entity set are *course_id*, *title*, *dept_name*, and *credits*.

In this section we consider only attributes that are **simple**— those not divided into subparts. In Section 6.3, we discuss more complex situations where attributes can be composite and multivalued.

Each entity has a **value** for each of its attributes. For instance, a particular *instructor* entity may have the value 12121 for *ID*, the value Wu for *name*, the value Finance for *dept_name*, and the value 90000 for *salary*.

The *ID* attribute is used to identify instructors uniquely, since there may be more than one instructor with the same name. Historically, many enterprises found it convenient to use a government-issued identification number as an attribute whose value uniquely identifies the person. However, that is considered bad practice for reasons of security and privacy. In general, the enterprise would have to create and assign its own unique identifier for each instructor.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. A database for a university may include a number of other entity sets. For example, in addition to keeping track of instructors and students, the university also has information about courses, which are represented by the entity set *course* with attributes *course_id*, *title*, *dept_name* and *credits*. In a real setting, a university database may keep dozens of entity sets.

An entity set is represented in an E-R diagram by a **rectangle**, which is divided into two parts. The first part, which in this text is shaded blue, contains the name of

Telegram: @uni_k

www.konkur.in

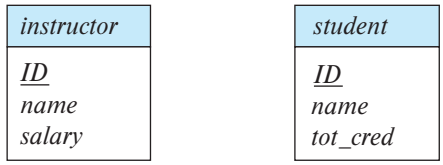


Figure 6.1 E-R diagram showing entity sets *instructor* and *student*.

the entity set. The second part contains the names of all the attributes of the entity set. The E-R diagram in Figure 6.1 shows two entity sets *instructor* and *student*. The attributes associated with *instructor* are *ID*, *name*, and *salary*. The attributes associated with *student* are *ID*, *name*, and *tot_cred*. Attributes that are part of the primary key are underlined (see Section 6.5).

6.2.2 Relationship Sets

A **relationship** is an association among several entities. For example, we can define a relationship *advisor* that associates instructor Katz with student Shankar. This relationship specifies that Katz is an advisor to student Shankar. A **relationship set** is a set of relationships of the same type.

Consider two entity sets *instructor* and *student*. We define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors. Figure 6.2 depicts this association. To keep the figure simple, only some of the attributes of the two entity sets are shown.

A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled. As an illustration, the individual *instructor* entity Katz, who has instructor *ID* 45565, and the *student* entity Shankar, who has student *ID* 12345, participate in a relationship instance of *advi-*

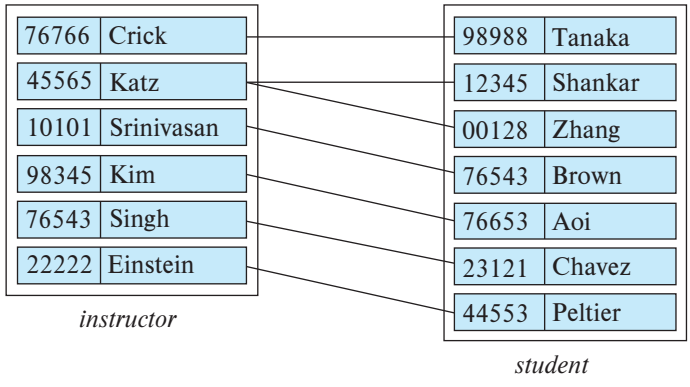


Figure 6.2 Relationship set *advisor* (only some attributes of *instructor* and *student* are shown).

Telegram: @uni_k

www.konkur.in

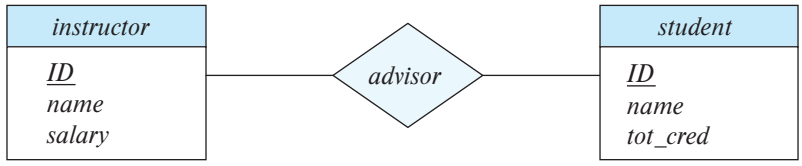


Figure 6.3 E-R diagram showing relationship set *advisor*.

sor. This relationship instance represents that in the university, the instructor Katz is advising student Shankar.

A relationship set is represented in an E-R diagram by a **diamond**, which is linked via **lines** to a number of different entity sets (rectangles). The E-R diagram in Figure 6.3 shows the two entity sets *instructor* and *student*, related through a binary relationship set *advisor*.

As another example, consider the two entity sets *student* and *section*, where *section* denotes an offering of a course. We can define the relationship set *takes* to denote the association between a student and a section in which that student is enrolled.

Although in the preceding examples each relationship set was an association between two entity sets, in general a relationship set may denote the association of more than two entity sets.

Formally, a **relationship set** is a mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where $(e_1, e_2, \ldots, e_n)$ is a relationship instance.

The association between entity sets is referred to as participation; i.e., the entity sets $E_1, E_2, \ldots, E_n$ **participate** in relationship set R.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification. Such is the case when the entity sets of a relationship set are not distinct; that is, the same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance. For example, consider the entity set *course* that records information about all the courses offered in the university. To depict the situation where one course (C2) is a prerequisite for another course (C1) we have relationship set *prereq* that is modeled by ordered pairs of *course* entities. The first course of a pair takes the role of course C1, whereas the second takes the role of prerequisite course C2. In this way, all relationships of *prereq* are characterized by (C1, C2) pairs; (C2, C1) pairs are excluded. We indicate roles in E-R diagrams by labeling the lines that connect diamonds

Telegram: @uni_k

www.konkur.in

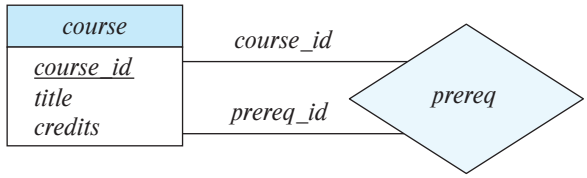


Figure 6.4 E-R diagram with role indicators.

to rectangles. Figure 6.4 shows the role indicators *course_id* and *prereq_id* between the *course* entity set and the *prereq* relationship set.

A relationship may also have attributes called **descriptive attributes**. As an example of descriptive attributes for relationships, consider the relationship set *takes* which relates entity sets *student* and *section*. We may wish to store a descriptive attribute *grade* with the relationship to record the grade that a student received in a course offering.

An attribute of a relationship set is represented in an E-R diagram by an **undivided rectangle**. We link the rectangle with a dashed line to the diamond representing that relationship set. For example, Figure 6.5 shows the relationship set *takes* between the entity sets *section* and *student*. We have the descriptive attribute *grade* attached to the relationship set *takes*. A relationship set may have multiple descriptive attributes; for example, we may also store a descriptive attribute *for_credit* with the *takes* relationship set to record whether a student has taken the section for credit, or is auditing (or sitting in on) the course.

Observe that the attributes of the two entity sets have been omitted from the E-R diagram in Figure 6.5, with the understanding that they are specified elsewhere in the complete E-R diagram for the university; we have already seen the attributes for *student*, and we will see the attributes of *section* later in this chapter. Complex E-R designs may need to be split into multiple diagrams that may be located in different pages. Relationship sets should be shown in only one location, but entity sets may be repeated in more than one location. The attributes of an entity set should be shown in the first occurrence. Subsequent occurrences of the entity set should be shown without attributes, to avoid repetition of information and the resultant possibility of inconsistency in the attributes shown in different occurrences.

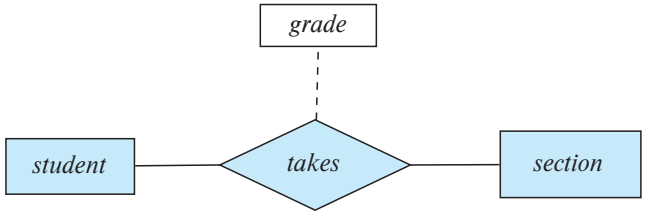


Figure 6.5 E-R diagram with an attribute attached to a relationship set.

Telegram: @uni_k

www.konkur.in

It is possible to have more than one relationship set involving the same entity sets. For example, suppose that students may be teaching assistants for a course. Then, the entity sets *section* and *student* may participate in a relationship set *teaching_assistant*, in addition to participating in the *takes* relationship set.

The formal definition of a relationship set, which we saw earlier, defines a relationship set as a set of relationship instances. Consider the *takes* relationship between *student* and *section*. Since a set cannot have duplicates, it follows that a particular student can have only one association with a particular section in the *takes* relationship. Thus, a student can have only one grade associated with a section, which makes sense in this case. However, if we wish to allow a student to have more than one grade for the same section, we need to have an attribute *grades* which stores a set of grades; such attributes are called multivalued attributes, and we shall see them later in Section 6.3.

The relationship sets *advisor* and *takes* provide examples of a **binary relationship set**—that is, one that involves two entity sets. Most of the relationship sets in a database system are binary. Occasionally, however, relationship sets involve more than two entity sets. The number of entity sets that participate in a relationship set is the **degree of the relationship set**. A binary relationship set is of degree 2; a **ternary relationship set** is of degree 3.

As an example, suppose that we have an entity set *project* that represents all the research projects carried out in the university. Consider the entity sets *instructor*, *student*, and *project*. Each project can have multiple associated students and multiple associated instructors. Furthermore, each student working on a project must have an associated instructor who guides the student on the project. For now, we ignore the first two relationships, between project and instructor, and between project and student. Instead, we focus on the information about which instructor is guiding which student on a particular project.

To represent this information, we relate the three entity sets through a ternary relationship set *proj_guide*, which relates entity sets *instructor*, *student*, and *project*. An instance of *proj_guide* indicates that a particular student is guided by a particular instructor on a particular project. Note that a student could have different instructors as guides for different projects, which cannot be captured by a binary relationship between students and instructors.

Nonbinary relationship sets can be specified easily in an E-R diagram. Figure 6.6 shows the E-R diagram representation of the ternary relationship set *proj_guide*.

6.3 Complex Attributes

For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute. The domain of attribute *course_id* might be the set of all text strings of a certain length. Similarly, the domain of attribute *semester* might be strings from the set {Fall, Winter, Spring, Summer}.

Telegram: @uni_k

www.konkur.in

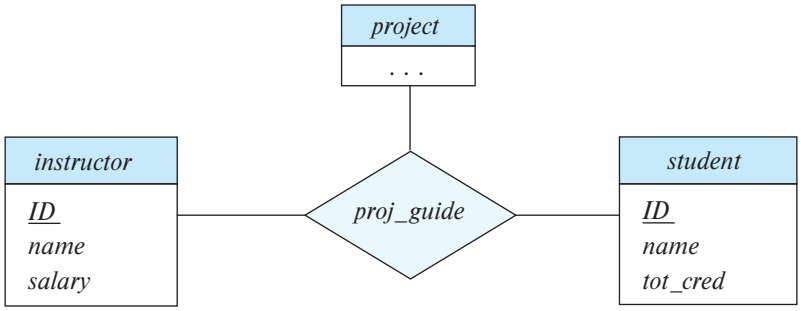


Figure 6.6 E-R diagram with a ternary relationship *proj_guide*.

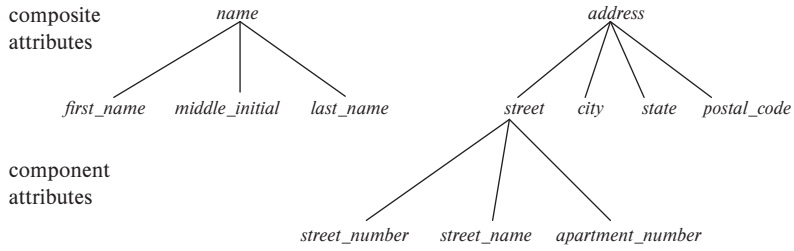


Figure 6.7 Composite attributes instructor *name* and *address*.

An attribute, as used in the E-R model, can be characterized by the following attribute types.

- **Simple** and **composite** attributes. In our examples thus far, the attributes have been **simple**; that is, they have not been divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (i.e., other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first_name*, *middle_initial*, and *last_name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Suppose we were to add an address to the *student* entity-set. The address can be defined as the composite attribute *address* with the attributes *street*, *city*, *state*, and *postal_code*.[1] Composite attributes help us to group together related attributes, making the modeling cleaner.

  Note also that a composite attribute may appear as a hierarchy. In the composite attribute *address*, its component attribute *street* can be further divided into *street_number*, *street_name*, and *apartment_number*. Figure 6.7 depicts these examples of composite attributes for the *instructor* entity set.

[1] We assume the address format used in the United States, which includes a

Telegram: @uni_k

www.konkur.in

- **Single-valued** and **multivalued** attributes. The attributes in our examples all have a single value for a particular entity. For instance, the *student_ID* attribute for a specific student entity refers to only one student *ID*. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Suppose we add to the *instructor* entity set a *phone_number* attribute. An *instructor* may have zero, one, or several phone numbers, and different instructors may have different numbers of phones. This type of attribute is said to be **multivalued**. As another example, we could add to the *instructor* entity set an attribute *dependent_name* listing all the dependents. This attribute would be multivalued, since any particular instructor may have zero, one, or more dependents.
- **Derived attributes**. The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the *instructor* entity set has an attribute *students_advised*, which represents how many students an instructor advises. We can derive the value for this attribute by counting the number of *student* entities associated with that instructor.

  As another example, suppose that the *instructor* entity set has an attribute *age* that indicates the instructor's age. If the *instructor* entity set also has an attribute *date_of_birth*, we can calculate *age* from *date_of_birth* and the current date. Thus, *age* is a derived attribute. In this case, *date_of_birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored but is computed when required.

Figure 6.8 shows how composite attributes can be represented in the E-R notation. Here, a composite attribute *name* with component attributes *first_name*, *middle_initial*, and *last_name* replaces the simple attribute *name* of *instructor*. As another example, suppose we were to add an address to the *instructor* entity set. The address can be defined as the composite attribute *address* with the attributes *street*, *city*, *state*, and *postal_code*. The attribute *street* is itself a composite attribute whose component attributes are *street_number*, *street_name*, and *apartment_number*. The figure also illustrates a multivalued attribute *phone_number*, denoted by "{*phone_number*}", and a derived attribute *age*, depicted by "*age* ( )".

An attribute takes a **null** value when an entity does not have a value for it. The *null* value may indicate "not applicable"—that is, the value does not exist for the entity. For example, a person who has no middle name may have the *middle_initial* attribute set to *null*. *Null* can also designate that an attribute value is unknown. An unknown value may be either *missing* (the value does exist, but we do not have that information) or *not known* (we do not know whether or not the value actually exists).

For instance, if the *name* value for a particular instructor is *null*, we assume that the value is missing, since every instructor must have a name. A null value for the *apartment_number* attribute could mean that the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what

Telegram: @uni_k

www.konkur.in

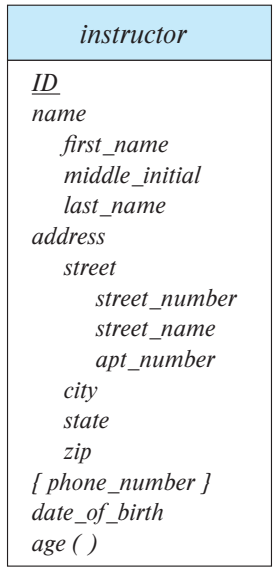


Figure 6.8 E-R diagram with composite, multivalued, and derived attributes.

it is (missing), or that we do not know whether or not an apartment number is part of the instructor's address (unknown).

6.4 Mapping Cardinalities

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

- **One-to-one**. An entity in A is associated with *at most* one entity in B, and an entity in B is associated with *at most* one entity in A. (See Figure 6.9a.)
- **One-to-many**. An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with *at most* one entity in A. (See Figure 6.9b.)
- **Many-to-one**. An entity in A is associated with *at most* one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A. (See Figure 6.10a.)

Telegram: @uni_k

www.konkur.in

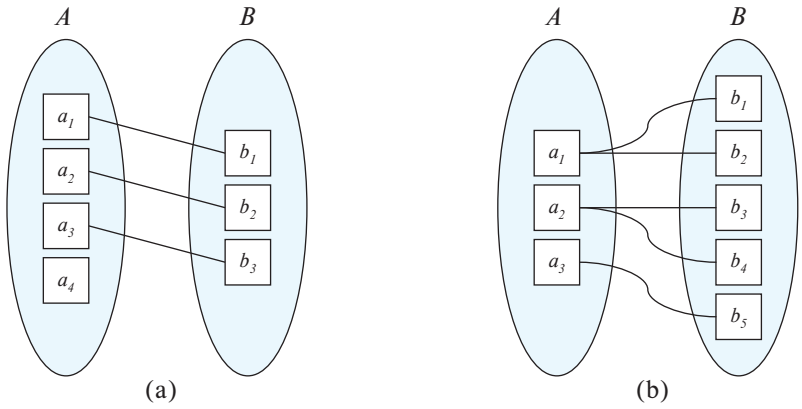


Figure 6.9 Mapping cardinalities. (a) One-to-one. (b) One-to-many.

- **Many-to-many**. An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A. (See Figure 6.10b.)

The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling.

As an illustration, consider the *advisor* relationship set. If a student can be advised by several instructors (as in the case of students advised jointly), the relationship set is many-to-many. In contrast, if a particular university imposes a constraint that a student can be advised by only one instructor, and an instructor can advise several students, then the relationship set from *instructor* to *student* must be one-to-many. Thus, mapping

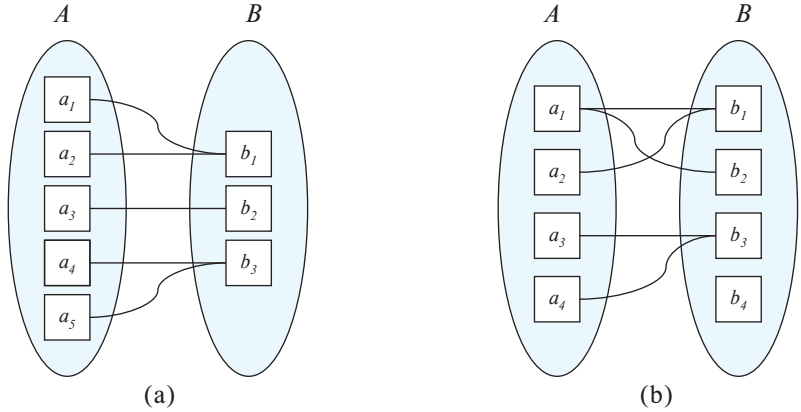


Figure 6.10 Mapping cardinalities. (a) Many-to-one. (b) Many-to-many.

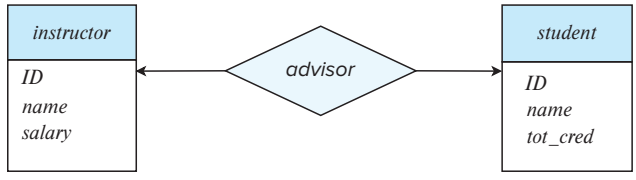
Telegram: @uni_k

www.konkur.in

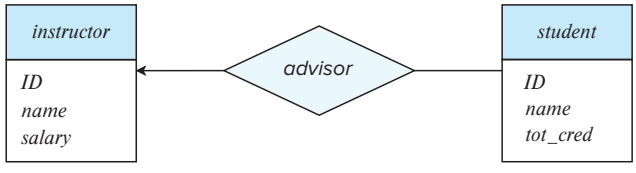
cardinalities can be used to specify constraints on what relationships are permitted in the real world.

In the E-R diagram notation, we indicate cardinality constraints on a relationship by drawing either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question. Specifically, for the university example:

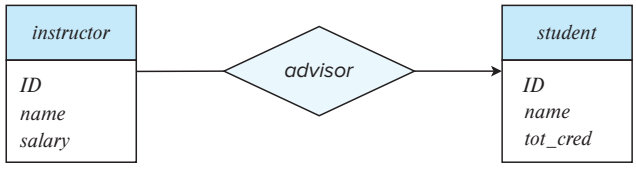
- **One-to-one**. We draw a directed line from the relationship set to both entity sets. For example, in Figure 6.11a, the directed lines to *instructor* and *student* indicate that an instructor may advise at most one student, and a student may have at most one advisor.



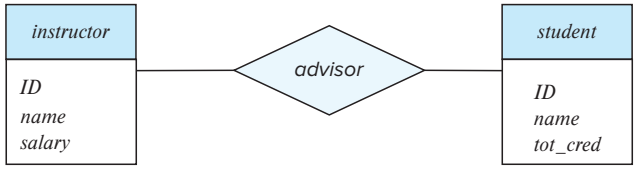
(a) One-to-one



(b) One-to-many



(c) Many-to-one



(d) Many-to-many

Figure 6.11 Relationship cardinalities.

Telegram: @uni_k

www.konkur.in

- **One-to-many**. We draw a directed line from the relationship set to the "one" side of the relationship. Thus, in Figure 6.11b, there is a directed line from relationship set *advisor* to the entity set *instructor*, and an undirected line to the entity set *student*. This indicates that an instructor may advise many students, but a student may have at most one advisor.
- **Many-to-one**. We draw a directed line from the relationship set to the "one" side of the relationship. Thus, in Figure 6.11c, there is an undirected line from the relationship set *advisor* to the entity set *instructor* and a directed line to the entity set *student*. This indicates that an instructor may advise at most one student, but a student may have many advisors.
- **Many-to-many**. We draw an undirected line from the relationship set to both entity sets. Thus, in Figure 6.11d, there are undirected lines from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise many students, and a student may have many advisors.

The participation of an entity set E in a relationship set R is said to be **total** if every entity in E must participate in at least one relationship in R. If it is possible that some entities in E do not participate in relationships in R, the participation of entity set E in relationship R is said to be **partial**.

For example, a university may require every *student* to have at least one advisor; in the E-R model, this corresponds to requiring each entity to be related to at least one instructor through the *advisor* relationship. Therefore, the participation of *student* in the relationship set *advisor* is total. In contrast, an *instructor* need not advise any students. Hence, it is possible that only some of the *instructor* entities are related to the *student* entity set through the *advisor* relationship, and the participation of *instructor* in the *advisor* relationship set is therefore partial.

We indicate total participation of an entity in a relationship set using double lines. Figure 6.12 shows an example of the *advisor* relationship set where the double line indicates that a student must have an advisor.

E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. A line may have an associated minimum and maximum cardinality, shown in the form $l..h$, where l

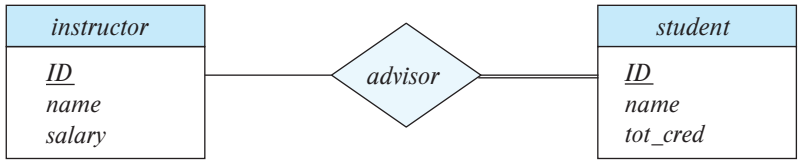


Figure 6.12 E-R diagram showing total participation.

Telegram: @uni_k

www.konkur.in

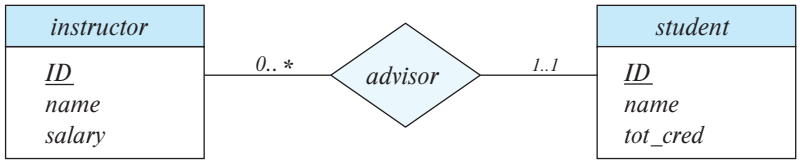


Figure 6.13 Cardinality limits on relationship sets.

is the minimum and h the maximum cardinality. A minimum value of 1 indicates total participation of the entity set in the relationship set; that is, each entity in the entity set occurs in at least one relationship in that relationship set. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value $*$ indicates no limit.

For example, consider Figure 6.13. The line between *advisor* and *student* has a cardinality constraint of 1..1, meaning the minimum and the maximum cardinality are both 1. That is, each student must have exactly one advisor. The limit 0.. $*$ on the line between *advisor* and *instructor* indicates that an instructor can have zero or more students. Thus, the relationship *advisor* is one-to-many from *instructor* to *student*, and further the participation of *student* in *advisor* is total, implying that a student must have an advisor.

It is easy to misinterpret the 0.. $*$ on the left edge and think that the relationship *advisor* is many-to-one from *instructor* to *student*—this is exactly the reverse of the correct interpretation.

If both edges have a maximum value of 1, the relationship is one-to-one. If we had specified a cardinality limit of 1.. $*$ on the left edge, we would be saying that each instructor must advise at least one student.

The E-R diagram in Figure 6.13 could alternatively have been drawn with a double line from *student* to *advisor*, and an arrow on the line from *advisor* to *instructor*, in place of the cardinality constraints shown. This alternative diagram would enforce exactly the same constraints as the constraints shown in the figure.

In the case of nonbinary relationship sets, we can specify some types of many-to-one relationships. Suppose a *student* can have at most one instructor as a guide on a project. This constraint can be specified by an arrow pointing to *instructor* on the edge from *proj_guide*.

We permit at most one arrow out of a nonbinary relationship set, since an E-R diagram with two or more arrows out of a nonbinary relationship set can be interpreted in two ways. We elaborate on this issue in Section 6.5.2.

6.5 Primary Key

We must have a way to specify how entities within a given entity set and relationships within a given relationship set are distinguished.

Telegram: @uni_k

www.konkur.in

6.5.1 Entity Sets

Conceptually, individual entities are distinct; from a database perspective, however, the differences among them must be expressed in terms of their attributes.

Therefore, the values of the attribute values of an entity must be such that they can *uniquely identify* the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.

The notion of a *key* for a relation schema, as defined in Section 2.3, applies directly to entity sets. That is, a key for an entity is a set of attributes that suffice to distinguish entities from each other. The concepts of superkey, candidate key, and primary key are applicable to entity sets just as they are applicable to relation schemas.

Keys also help to identify relationships uniquely, and thus distinguish relationships from each other. Next, we define the corresponding notions of keys for relationship sets.

6.5.2 Relationship Sets

We need a mechanism to distinguish among the various relationships of a relationship set.

Let R be a relationship set involving entity sets $E_1, E_2, \ldots, E_n$. Let *primary-key*(E_i) denote the set of attributes that forms the primary key for entity set E_i. Assume for now that the attribute names of all primary keys are unique. The composition of the primary key for a relationship set depends on the set of attributes associated with the relationship set R.

If the relationship set R has no attributes associated with it, then the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n)$$

describes an individual relationship in set R.

If the relationship set R has attributes $a_1, a_2, \ldots, a_m$ associated with it, then the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n) \cup \{a_1, a_2, \ldots, a_m\}$$

describes an individual relationship in set R.

If the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them; the name of the entity set combined with the name of the attribute would form a unique name. If an entity set participates more than once in a relationship set (as in the *prereq* relationship in Section 6.2.2), the role name is used instead of the name of the entity set, to form a unique attribute name.

Recall that a relationship set is a set of relationship instances, and each instance is uniquely identified by the entities that participate in it. Thus, in both of the preceding cases, the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n)$$

forms a superkey for the relationship set.

Telegram: @uni_k

www.konkur.in

The choice of the primary key for a binary relationship set depends on the mapping cardinality of the relationship set. For many-to-many relationships, the preceding union of the primary keys is a minimal superkey and is chosen as the primary key. As an illustration, consider the entity sets *instructor* and *student*, and the relationship set *advisor*, in Section 6.2.2. Suppose that the relationship set is many-to-many. Then the primary key of *advisor* consists of the union of the primary keys of *instructor* and *student*.

For one-to-many and many-to-one relationships, the primary key of the "many" side is a minimal superkey and is used as the primary key. For example, if the relationship is many-to-one from *student* to *instructor*—that is, each student can have at most one advisor—then the primary key of *advisor* is simply the primary key of *student*. However, if an instructor can advise only one student—that is, if the *advisor* relationship is many-to-one from *instructor* to *student*—then the primary key of *advisor* is simply the primary key of *instructor*.

For one-to-one relationships, the primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key of the relationship set. However, if an instructor can advise only one student, and each student can be advised by only one instructor—that is, if the *advisor* relationship is one-to-one—then the primary key of either *student* or *instructor* can be chosen as the primary key for *advisor*.

For nonbinary relationships, if no cardinality constraints are present, then the superkey formed as described earlier in this section is the only candidate key, and it is chosen as the primary key. The choice of the primary key is more complicated if cardinality constraints are present. As we noted in Section 6.4, we permit at most one arrow out of a relationship set. We do so because an E-R diagram with two or more arrows out of a nonbinary relationship set can be interpreted in the two ways we describe below.

Suppose there is a relationship set R between entity sets E_1, E_2, E_3, E_4, and the only arrows are on the edges to entity sets E_3 and E_4. Then, the two possible interpretations are:

1. A particular combination of entities from E_1, E_2 can be associated with at most one combination of entities from E_3, E_4. Thus, the primary key for the relationship R can be constructed by the union of the primary keys of E_1 and E_2.
2. A particular combination of entities from E_1, E_2, E_3 can be associated with at most one combination of entities from E_4, and further a particular combination of entities from E_1, E_2, E_4 can be associated with at most one combination of entities from E_3, Then the union of the primary keys of E_1, E_2, and E_3 forms a candidate key, as does the union of the primary keys of E_1, E_2, and E_4.

Each of these interpretations has been used in practice and both are correct for particular enterprises being modeled. Thus, to avoid confusion, we permit only one arrow out of a nonbinary relationship set, in which case the two interpretations are equivalent.

Telegram: @uni_k

www.konkur.in

In order to represent a situation where one of the multiple-arrow situations holds, the E-R design can be modified by replacing the non-binary relationship set with an entity set. That is, we treat each instance of the non-binary relationship set as an entity. Then we can relate each of those entities to corresponding instances of E_1, E_2, E_4 via separate relationship sets. A simpler approach is to use *functional dependencies*, which we study in Chapter 7 (Section 7.4). Functional dependencies which allow either of these interpretations to be specified simply in an unambiguous manner.

The primary key for the relationship set R is then the union of the primary keys of those participating entity sets E_i that do not have an incoming arrow from the relationship set R.

6.5.3 Weak Entity Sets

Consider a *section* entity, which is uniquely identified by a course identifier, semester, year, and section identifier. Section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.

Now, observe that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related. One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *sec_id*, *year*, and *semester*.[2] However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely; although each *section* entity is distinct, sections for different courses may share the same *sec_id*, *year*, and *semester*. To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case the *course_id*, required to identify *section* entities uniquely.

The notion of *weak entity set* formalizes the above intuition. A **weak entity set** is one whose existence is dependent on another entity set, called its **identifying entity set**; instead of associating a primary key with a weak entity, we use the primary key of the identifying entity, along with extra attributes, called **discriminator attributes** to uniquely identify a weak entity. An entity set that is not a weak entity set is termed a **strong entity set**.

Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.

The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

[2]Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped

Telegram: @uni_k

www.konkur.in

The identifying relationship set should not have any descriptive attributes, since any such attributes can instead be associated with the weak entity set.

In our example, the identifying entity set for *section* is *course*, and the relationship *sec_course*, which associates *section* entities with their corresponding *course* entities, is the identifying relationship. The primary key of *section* is formed by the primary key of the identifying entity set (that is, *course*), plus the discriminator of the weak entity set (that is, *section*). Thus, the primary key is {*course_id*, *sec_id*, *year*, *semester*}.

Note that we could have chosen to make *sec_id* globally unique across all courses offered in the university, in which case the *section* entity set would have had a primary key. However, conceptually, a *section* is still dependent on a *course* for its existence, which is made explicit by making it a weak entity set.

In E-R diagrams, a weak entity set is depicted via a double rectangle with the discriminator being underlined with a dashed line. The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond. In Figure 6.14, the weak entity set *section* depends on the strong entity set *course* via the relationship set *sec_course*.

The figure also illustrates the use of double lines to indicate that the participation of the (weak) entity set *section* in the relationship *sec_course* is *total*, meaning that every section must be related via *sec_course* to some course. Finally, the arrow from *sec_course* to *course* indicates that each section is related to a single course.

In general, a weak entity set must have a total participation in its identifying relationship set, and the relationship is many-to-one toward the identifying entity set.

A weak entity set can participate in relationships other than the identifying relationship. For instance, the *section* entity could participate in a relationship with the *time_slot* entity set, identifying the time when a particular class section meets. A weak entity set may participate as owner in an identifying relationship with another weak entity set. It is also possible to have a weak entity set with more than one identifying entity set. A particular weak entity would then be identified by a combination of entities, one from each identifying entity set. The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set.

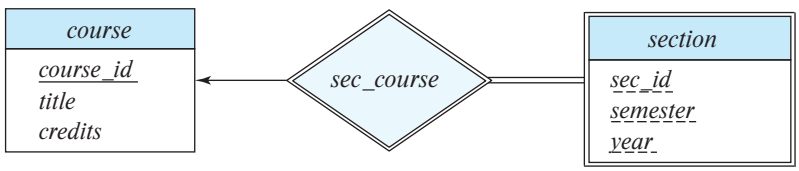


Figure 6.14 E-R diagram with a weak entity set.

Telegram: @uni_k

www.konkur.in

6.6 Removing Redundant Attributes in Entity Sets

When we design a database using the E-R model, we usually start by identifying those entity sets that should be included. For example, in the university organization we have discussed thus far, we decided to include such entity sets as *student* and *instructor*. Once the entity sets are decided upon, we must choose the appropriate attributes. These attributes are supposed to represent the various values we want to capture in the database. In the university organization, we decided that for the *instructor* entity set, we will include the attributes *ID*, *name*, *dept_name*, and *salary*. We could have added the attributes *phone_number*, *office_number*, *home_page*, and others. The choice of what attributes to include is up to the designer, who has a good understanding of the structure of the enterprise.

Once the entities and their corresponding attributes are chosen, the relationship sets among the various entities are formed. These relationship sets may result in a situation where attributes in the various entity sets are redundant and need to be removed from the original entity sets. To illustrate, consider the entity sets *instructor* and *department*:

- The entity set *instructor* includes the attributes *ID*, *name*, *dept_name*, and *salary*, with *ID* forming the primary key.
- The entity set *department* includes the attributes *dept_name*, *building*, and *budget*, with *dept_name* forming the primary key.

We model the fact that each instructor has an associated department using a relationship set *inst_dept* relating *instructor* and *department*.

The attribute *dept_name* appears in both entity sets. Since it is the primary key for the entity set *department*, it is redundant in the entity set *instructor* and needs to be removed.

Removing the attribute *dept_name* from the *instructor* entity set may appear rather unintuitive, since the relation *instructor* that we used in the earlier chapters had an attribute *dept_name*. As we shall see later, when we create a relational schema from the E-R diagram, the attribute *dept_name* in fact gets added to the relation *instructor*, but only if each instructor has at most one associated department. If an instructor has more than one associated department, the relationship between instructors and departments is recorded in a separate relation *inst_dept*.

Treating the connection between instructors and departments uniformly as a relationship, rather than as an attribute of *instructor*, makes the logical relationship explicit, and it helps avoid a premature assumption that each instructor is associated with only one department.

Similarly, the *student* entity set is related to the *department* entity set through the relationship set *student_dept* and thus there is no need for a *dept_name* attribute in *student*.

Telegram: @uni_k

www.konkur.in

As another example, consider course offerings (sections) along with the time slots of the offerings. Each time slot is identified by a *time_slot_id*, and has associated with it a set of weekly meetings, each identified by a day of the week, start time, and end time. We decide to model the set of weekly meeting times as a multivalued composite attribute. Suppose we model entity sets *section* and *time_slot* as follows:

- The entity set *section* includes the attributes *course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, and *time_slot_id*, with (*course_id*, *sec_id*, *year*, *semester*) forming the primary key.
- The entity set *time_slot* includes the attributes *time_slot_id*, which is the primary key,[3] and a multivalued composite attribute {(*day*, *start_time*, *end_time*)}.[4]

These entities are related through the relationship set *sec_time_slot*.

The attribute *time_slot_id* appears in both entity sets. Since it is the primary key for the entity set *time_slot*, it is redundant in the entity set *section* and needs to be removed.

As a final example, suppose we have an entity set *classroom*, with attributes *building*, *room_number*, and *capacity*, with *building* and *room_number* forming the primary key. Suppose also that we have a relationship set *sec_class* that relates *section* to *classroom*. Then the attributes {*building*, *room_number*} are redundant in the entity set *section*.

A good entity-relationship design does not contain redundant attributes. For our university example, we list the entity sets and their attributes below, with primary keys underlined:

- *classroom*: with attributes (*building*, *room_number*, *capacity*).
- *department*: with attributes (*dept_name*, *building*, *budget*).
- *course*: with attributes (*course_id*, *title*, *credits*).
- *instructor*: with attributes (*ID*, *name*, *salary*).
- *section*: with attributes (*course_id*, *sec_id*, *semester*, *year*).
- *student*: with attributes (*ID*, *name*, *tot_cred*).
- *time_slot*: with attributes (*time_slot_id*, {(*day*, *start_time*, *end_time*) }).

The relationship sets in our design are listed below:

- *inst_dept*: relating instructors with departments.
- *stud_dept*: relating students with departments.

[3] We shall see later on that the primary key for the relation created from the entity set *time_slot* includes *day* and *start_time*; however, *day* and *start_time* do not form part of the primary key of the entity set *time_slot*.

[4] We could optionally give a name, such as *meeting*, for the composite

Telegram: @uni_k

www.konkur.in

- *teaches*: relating instructors with sections.
- *takes*: relating students with sections, with a descriptive attribute *grade*.
- *course_dept*: relating courses with departments.
- *sec_course*: relating sections with courses.
- *sec_class*: relating sections with classrooms.
- *sec_time_slot*: relating sections with time slots.
- *advisor*: relating students with instructors.
- *prereq*: relating courses with prerequisite courses.

You can verify that none of the entity sets has any attribute that is made redundant by one of the relationship sets. Further, you can verify that all the information (other than constraints) in the relational schema for our university database, which we saw earlier in Figure 2.9, has been captured by the above design, but with several attributes in the relational design replaced by relationships in the E-R design.

We are finally in a position to show (Figure 6.15) the E-R diagram that corresponds to the university enterprise that we have been using thus far in the text. This E-R diagram is equivalent to the textual description of the university E-R model, but with several additional constraints.

In our university database, we have a constraint that each instructor must have exactly one associated department. As a result, there is a double line in Figure 6.15 between *instructor* and *inst_dept*, indicating total participation of *instructor* in *inst_dept*; that is, each instructor must be associated with a department. Further, there is an arrow from *inst_dept* to *department*, indicating that each instructor can have at most one associated department.

Similarly, entity set *course* has a double line to relationship set *course_dept*, indicating that every course must be in some department, and entity set *student* has a double line to relationship set *stud_dept*, indicating that every student must be majoring in some department. In each case, an arrow points to the entity set *department* to show that a course (and, respectively, a student) can be related to only one department, not several.

Similarly, entity set *course* has a double line to relationship set *course_dept*, indicating that every course must be in some department, and entity set *student* has a double line to relationship set *stud_dept*, indicating that every student must be majoring in some department. In each case, an arrow points to the entity set *department* to show that a course (and, respectively, a student) can be related to only one department, not several.

Further, Figure 6.15 shows that the relationship set *takes* has a descriptive attribute *grade*, and that each student has at most one advisor. The figure also shows that *section* is a weak entity set, with attributes *sec_id*, *semester*, and *year* forming the discriminator; *sec_course* is the identifying relationship set relating weak entity set *section* to the strong entity set *course*.

Telegram: @uni_k

www.konkur.in

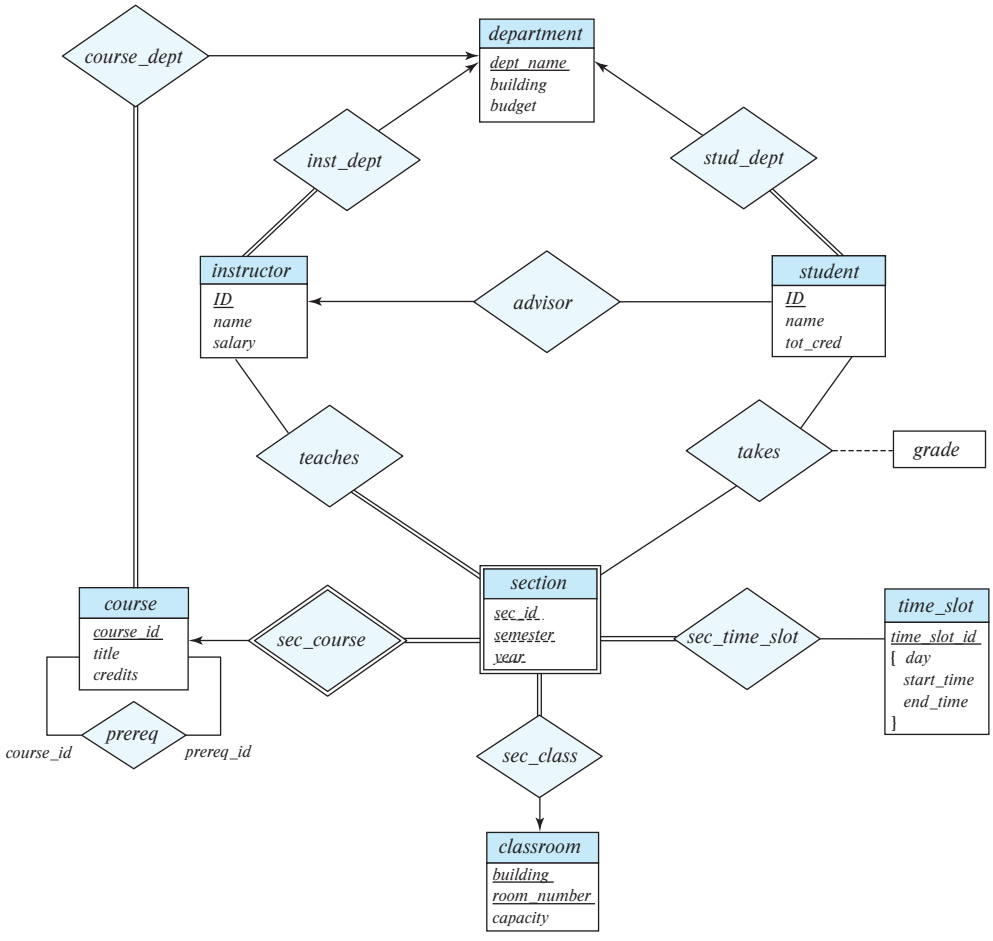


Figure 6.15 E-R diagram for a university enterprise.

In Section 6.7, we show how this E-R diagram can be used to derive the various relation schemas we use.

6.7 Reducing E-R Diagrams to Relational Schemas

Both the E-R model and the relational database model are abstract, logical representations of real-world enterprises. Because the two models employ similar design principles, we can convert an E-R design into a relational design. For each entity set and for each relationship set in the database design, there is a unique relation schema to which we assign the name of the corresponding entity set or relationship set.

Telegram: @uni_k

www.konkur.in

In this section, we describe how an E-R schema can be represented by relation schemas and how constraints arising from the E-R design can be mapped to constraints on relation schemas.

6.7.1 Representation of Strong Entity Sets

Let E be a strong entity set with only simple descriptive attributes $a_1, a_2, \ldots, a_n$. We represent this entity with a schema called E with n distinct attributes. Each tuple in a relation on this schema corresponds to one entity of the entity set E.

For schemas derived from strong entity sets, the primary key of the entity set serves as the primary key of the resulting schema. This follows directly from the fact that each tuple corresponds to a specific entity in the entity set.

As an illustration, consider the entity set *student* of the E-R diagram in Figure 6.15. This entity set has three attributes: *ID*, *name*, *tot_cred*. We represent this entity set by a schema called *student* with three attributes:

$$student\ (\underline{ID}, name, tot_cred)$$

Note that since student *ID* is the primary key of the entity set, it is also the primary key of the relation schema.

Continuing with our example, for the E-R diagram in Figure 6.15, all the strong entity sets, except *time_slot*, have only simple attributes. The schemas derived from these strong entity sets are depicted in Figure 6.16. Note that the *instructor*, *student*, and *course* schemas are different from the schemas we have used in the previous chapters (they do not contain the attribute *dept_name*). We shall revisit this issue shortly.

6.7.2 Representation of Strong Entity Sets with Complex Attributes

When a strong entity set has nonsimple attributes, things are a bit more complex. We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate attribute for the composite attribute itself. To illustrate, consider the version of the *instructor* entity set depicted in Figure 6.8. For the composite attribute *name*, the schema generated for *instructor* contains the attributes

---

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *credits*)
instructor(*ID*, *name*, *salary*)
student(*ID*, *name*, *tot_cred*)

Figure 6.16 Schemas derived from the entity sets in the E-R diagram in Figure 6.15.

Telegram: @uni_k

www.konkur.in

first_name, *middle_initial*, and *last_name*; there is no separate attribute or schema for *name*. Similarly, for the composite attribute *address*, the schema generated contains the attributes *street*, *city*, *state*, and *postal_code*. Since *street* is a composite attribute it is replaced by *street_number*, *street_name*, and *apt_number*.

Multivalued attributes are treated differently from other attributes. We have seen that attributes in an E-R diagram generally map directly into attributes for the appropriate relation schemas. Multivalued attributes, however, are an exception; new relation schemas are created for these attributes, as we shall see shortly.

Derived attributes are not explicitly represented in the relational data model. However, they can be represented as stored procedures, functions, or methods in other data models.

The relational schema derived from the version of entity set *instructor* with complex attributes, without including the multivalued attribute, is thus:

instructor (*ID*, *first_name*, *middle_initial*, *last_name*,
street_number, *street_name*, *apt_number*,
city, *state*, *postal_code*, *date_of_birth*)

For a multivalued attribute M, we create a relation schema R with an attribute A that corresponds to M and attributes corresponding to the primary key of the entity set or relationship set of which M is an attribute.

As an illustration, consider the E-R diagram in Figure 6.8 that depicts the entity set *instructor*, which includes the multivalued attribute *phone_number*. The primary key of *instructor* is *ID*. For this multivalued attribute, we create a relation schema

instructor_phone (*ID*, *phone_number*)

Each phone number of an instructor is represented as a unique tuple in the relation on this schema. Thus, if we had an instructor with *ID* 22222, and phone numbers 555-1234 and 555-4321, the relation *instructor_phone* would have two tuples (22222, 555-1234) and (22222, 555-4321).

We create a primary key of the relation schema consisting of all attributes of the schema. In the above example, the primary key consists of both attributes of the relation schema *instructor_phone*.

In addition, we create a foreign-key constraint on the relation schema created from the multivalued attribute. In that newly created schema, the attribute generated from the primary key of the entity set must reference the relation generated from the entity set. In the above example, the foreign-key constraint on the *instructor_phone* relation would be that attribute *ID* references the *instructor* relation.

In the case that an entity set consists of only two attributes—a single primary-key attribute B and a single multivalued attribute M— the relation schema for the entity set would contain only one attribute, namely, the primary-key attribute B. We can drop

Telegram: @uni_k

www.konkur.in

this relation, while retaining the relation schema with the attribute B and attribute A that corresponds to M.

To illustrate, consider the entity set *time_slot* depicted in Figure 6.15. Here, *time_slot_id* is the primary key of the *time_slot* entity set, and there is a single multivalued attribute that happens also to be composite. The entity set can be represented by just the following schema created from the multivalued composite attribute:

time_slot (*time_slot_id*, *day*, *start_time*, *end_time*)

Although not represented as a constraint on the E-R diagram, we know that there cannot be two meetings of a class that start at the same time of the same day of the week but end at different times; based on this constraint, *end_time* has been omitted from the primary key of the *time_slot* schema.

The relation created from the entity set would have only a single attribute *time_slot_id*; the optimization of dropping this relation has the benefit of simplifying the resultant database schema, although it has a drawback related to foreign keys, which we briefly discuss in Section 6.7.4.

6.7.3 Representation of Weak Entity Sets

Let A be a weak entity set with attributes $a_1, a_2, \ldots, a_m$. Let B be the strong entity set on which A depends. Let the primary key of B consist of attributes $b_1, b_2, \ldots, b_n$. We represent the entity set A by a relation schema called A with one attribute for each member of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

For schemas derived from a weak entity set, the combination of the primary key of the strong entity set and the discriminator of the weak entity set serves as the primary key of the schema. In addition to creating a primary key, we also create a foreign-key constraint on the relation A, specifying that the attributes $b_1, b_2, \ldots, b_n$ reference the primary key of the relation B. The foreign-key constraint ensures that for each tuple representing a weak entity, there is a corresponding tuple representing the corresponding strong entity.

As an illustration, consider the weak entity set *section* in the E-R diagram of Figure 6.15. This entity set has the attributes: *sec_id*, *semester*, and *year*. The primary key of the *course* entity set, on which *section* depends, is *course_id*. Thus, we represent *section* by a schema with the following attributes:

section (*course_id*, *sec_id*, *semester*, *year*)

The primary key consists of the primary key of the entity set *course*, along with the discriminator of *section*, which is *sec_id*, *semester*, and *year*. We also create a foreign-key

Telegram: @uni_k

www.konkur.in

constraint on the *section* schema, with the attribute *course_id* referencing the primary key of the *course* schema.[5]

6.7.4 Representation of Relationship Sets

Let R be a relationship set, let $a_1, a_2, \ldots, a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in R, and let the descriptive attributes (if any) of R be $b_1, b_2, \ldots, b_n$. We represent this relationship set by a relation schema called R with one attribute for each member of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

We described in Section 6.5, how to choose a primary key for a binary relationship set. The primary key attributes of the relationship set are also used as the primary key attributes of the relational schema R.

As an illustration, consider the relationship set *advisor* in the E-R diagram of Figure 6.15. This relationship set involves the following entity sets:

- *instructor*, with the primary key *ID*.
- *student*, with the primary key *ID*.

Since the relationship set has no attributes, the *advisor* schema has two attributes, the primary keys of *instructor* and *student*. Since both attributes have the same name, we rename them *i_ID* and s_*ID*. Since the *advisor* relationship set is many-to-one from *student* to *instructor* the primary key for the *advisor* relation schema is *s_ID*.

We also create foreign-key constraints on the relation schema R as follows: For each entity set E_i related by relationship set R, we create a foreign-key constraint from relation schema R, with the attributes of R that were derived from primary-key attributes of E_i referencing the primary key of the relation schema representing E_i.

Returning to our earlier example, we thus create two foreign-key constraints on the *advisor* relation, with attribute *i_ID* referencing the primary key of *instructor* and attribute *s_ID* referencing the primary key of *student*.

Applying the preceding techniques to the other relationship sets in the E-R diagram in Figure 6.15, we get the relational schemas depicted in Figure 6.17.

Observe that for the case of the relationship set *prereq*, the role indicators associated with the relationship are used as attribute names, since both roles refer to the same relation *course*.

Similar to the case of *advisor*, the primary key for each of the relations *sec_course*, *sec_time_slot*, *sec_class*, *inst_dept*, *stud_dept*, and *course_dept* consists of the primary key

[5]Optionally, the foreign-key constraint could have an "on delete cascade" specification, so that deletion of a *course* entity automatically deletes any *section* entities that reference the *course* entity. Without that specification, each section of a course would have to be deleted before the corresponding course can be

Telegram: @uni_k

www.konkur.in

teaches (*ID*, *course_id*, *sec_id*, *semester*, *year*)
takes (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
prereq (*course_id*, *prereq_id*)
advisor (*s_ID*, *i_ID*)
sec_course (*course_id*, *sec_id*, *semester*, *year*)
sec_time_slot (*course_id*, *sec_id*, *semester*, *year*, *time_slot_id*)
sec_class (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*)
inst_dept (*ID*, *dept_name*)
stud_dept (*ID*, *dept_name*)
course_dept (*course_id*, *dept_name*)

Figure 6.17 Schemas derived from relationship sets in the E-R diagram in Figure 6.15.

of only one of the two related entity sets, since each of the corresponding relationships is many-to-one.

Foreign keys are not shown in Figure 6.17, but for each of the relations in the figure there are two foreign-key constraints, referencing the two relations created from the two related entity sets. Thus, for example, *sec_course* has foreign keys referencing *section* and *classroom*, *teaches* has foreign keys referencing *instructor* and *section*, and *takes* has foreign keys referencing *student* and *section*.

The optimization that allowed us to create only a single relation schema from the entity set *time_slot*, which had a multivalued attribute, prevents the creation of a foreign key from the relation schema *sec_time_slot* to the relation created from entity set *time_slot*, since we dropped the relation created from the entity set *time_slot*. We retained the relation created from the multivalued attribute and named it *time_slot*, but this relation may potentially have no tuples corresponding to a *time_slot_id*, or it may have multiple tuples corresponding to a *time_slot_id*; thus, *time_slot_id* in *sec_time_slot* cannot reference this relation.

The astute reader may wonder why we have not seen the schemas *sec_course*, *sec_time_slot*, *sec_class*, *inst_dept*, *stud_dept*, and *course_dept* in the previous chapters. The reason is that the algorithm we have presented thus far results in some schemas that can be either eliminated or combined with other schemas. We explore this issue next.

6.7.5 Redundancy of Schemas

A relationship set linking a weak entity set to the corresponding strong entity set is treated specially. As we noted in Section 6.5.3, these relationships are many-to-one and have no descriptive attributes. Furthermore, the of a weak entity set includes the primary key of the strong entity set. In the E-R diagram of Figure 6.14, the weak entity set *section* is dependent on the strong entity set *course* via the relationship set *sec course*.

Telegram: @uni_k

www.konkur.in

The primary key of *section* is {*course_id*, *sec_id*, *semester*, *year*}, and the primary key of *course* is *course_id*. Since *sec_course* has no descriptive attributes, the *sec_course* schema has attributes *course_id*, *sec_id*, *semester*, and *year*. The schema for the entity set *section* includes the attributes *course_id*, *sec_id*, *semester*, and *year* (among others). Every (*course_id*, *sec_id*, *semester*, *year*) combination in a *sec_course* relation would also be present in the relation on schema *section*, and vice versa. Thus, the *sec_course* schema is redundant.

In general, the schema for the relationship set linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a relational database design based upon an E-R diagram.

6.7.6 Combination of Schemas

Consider a many-to-one relationship set *AB* from entity set *A* to entity set *B*. Using our relational-schema construction algorithm outlined previously, we get three schemas: *A*, *B*, and *AB*. Suppose further that the participation of *A* in the relationship is total; that is, every entity *a* in the entity set *A* must participate in the relationship *AB*. Then we can combine the schemas *A* and *AB* to form a single schema consisting of the union of attributes of both schemas. The primary key of the combined schema is the primary key of the entity set into whose schema the relationship set schema was merged.

To illustrate, let's examine the various relations in the E-R diagram of Figure 6.15 that satisfy the preceding criteria:

- *inst_dept*. The schemas *instructor* and *department* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *inst_dept* can be combined with the *instructor* schema. The resulting *instructor* schema consists of the attributes {*ID*, *name*, *dept_name*, *salary*}.
- *stud_dept*. The schemas *student* and *department* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *stud_dept* can be combined with the *student* schema. The resulting *student* schema consists of the attributes {*ID*, *name*, *dept_name*, *tot_cred*}.
- *course_dept*. The schemas *course* and *department* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *course_dept* can be combined with the *course* schema. The resulting *course* schema consists of the attributes {*course_id*, *title*, *dept_name*, *credits*}.
- *sec_class*. The schemas *section* and *classroom* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *sec_class* can be combined with the *section* schema. The resulting *section* schema consists of the attributes {*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*}.
- *sec_time_slot*. The schemas *section* and *time_slot* correspond to the entity sets *A* and *B* respectively, Thus, the schema *sec_time_slot* can be combined with the *section*

Telegram: @uni_k

www.konkur.in

schema obtained in the previous step. The resulting *section* schema consists of the attributes {*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*}.

In the case of one-to-one relationships, the relation schema for the relationship set can be combined with the schemas for either of the entity sets.

We can combine schemas even if the participation is partial by using null values. In the preceding example, if *inst_dept* were partial, then we would store null values for the *dept_name* attribute for those instructors who have no associated department.

Finally, we consider the foreign-key constraints that would have appeared in the schema representing the relationship set. There would have been foreign-key constraints referencing each of the entity sets participating in the relationship set. We drop the constraint referencing the entity set into whose schema the relationship set schema is merged, and add the other foreign-key constraints to the combined schema. For example, *inst_dept* has a foreign key constraint of the attribute *dept_name* referencing the *department* relation. This foreign constraint is enforced implicitly by the *instructor* relation when the schema for *inst_dept* is merged into *instructor*.

6.8 Extended E-R Features

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. In this section, we discuss the extended E-R features of specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation.

To help with the discussions, we shall use a slightly more elaborate database schema for the university. In particular, we shall model the various people within a university by defining an entity set *person*, with attributes *ID*, *name*, *street*, and *city*.

6.8.1 Specialization

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings.

As an example, the entity set *person* may be further classified as one of the following:

- *employee*.
- *student*.

Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For example, *employee* entities may be described further by the attribute *salary*, whereas *student* entities may

Telegram: @uni_k

www.konkur.in

be described further by the attribute *tot_cred*. The process of designating subgroupings within an entity set is called **specialization**. The specialization of *person* allows us to distinguish among person entities according to whether they correspond to employees or students: in general, a person could be an employee, a student, both, or neither.

As another example, suppose the university divides students into two categories: graduate and undergraduate. Graduate students have an office assigned to them. Undergraduate students are assigned to a residential college. Each of these student types is described by a set of attributes that includes all the attributes of the entity set *student* plus additional attributes.

We can apply specialization repeatedly to refine a design. The university could create two specializations of *student*, namely *graduate* and *undergraduate*. As we saw earlier, student entities are described by the attributes *ID*, *name*, *street, city*, and *tot_cred*. The entity set *graduate* would have all the attributes of *student* and an additional attribute *office_number*. The entity set *undergraduate* would have all the attributes of *student*, and an additional attribute *residential_college*. As another example, university employees may be further classified as one of *instructor* or *secretary*.

Each of these employee types is described by a set of attributes that includes all the attributes of entity set *employee* plus additional attributes. For example, *instructor* entities may be described further by the attribute *rank* while *secretary* entities are described by the attribute *hours_per_week*. Further, *secretary* entities may participate in a relationship *secretary_for* between the *secretary* and *employee* entity sets, which identifies the employees who are assisted by a secretary.

An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs. Another, coexistent, specialization could be based on whether the person is a temporary (limited_term) employee or a permanent employee, resulting in the entity sets *temporary_employee* and *permanent_employee*. When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations. For instance, a given employee may be a temporary employee who is a secretary.

In terms of an E-R diagram, specialization is depicted by a hollow arrow-head pointing from the specialized entity to the other entity (see Figure 6.18). We refer to this relationship as the ISA relationship, which stands for "is a" and represents, for example, that an instructor "is a" employee.

The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**. For an overlapping specialization (as is the case for *student* and *employee* as specializations of *person*), two separate arrows are used. For a disjoint specialization (as is the case for *instructor* and *secretary* as specializations of *employee*), a single arrow is used. The specialization relationship may also be referred to as a **superclass-subclass** relationship. Higher- and lower-level entity sets are depicted as regular entity sets—that is, as rectangles containing the name of the entity set.

Telegram: @uni_k

www.konkur.in

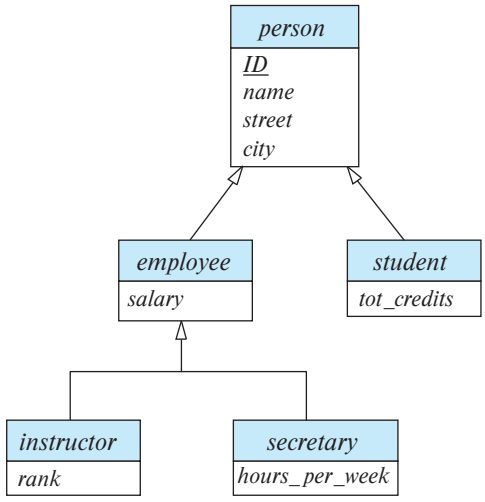


Figure 6.18 Specialization and generalization.

6.8.2 Generalization

The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features. The database designer may have first identified:

- *instructor* entity set with attributes *instructor_id*, *instructor_name*, *instructor_salary*, and *rank*.
- *secretary* entity set with attributes *secretary_id*, *secretary_name*, *secretary_salary*, and *hours_per_week*.

There are similarities between the *instructor* entity set and the *secretary* entity set in the sense that they have several attributes that are conceptually the same across the two entity sets: namely, the identifier, name, and salary attributes. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *employee* is the higher-level entity set and *instructor* and *secretary* are lower-level entity sets. In this case, attributes that are conceptually the same had different names in the two lower-level entity sets. To create a generalization, the attributes must be given a common name and represented with the higher-level entity *person*. We can use the attribute names *ID*, *name*, *street*, and *city*, as we saw in the example in Section 6.8.1.

Telegram: @uni_k

www.konkur.in

Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *employee* and *student* subclasses.

For all practical purposes, generalization is a simple inversion of specialization. We apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation are distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

Specialization stems from a single entity set; it emphasizes differences among entities within the set by creating distinct lower-level entity sets. These lower-level entity sets may have attributes, or may participate in relationships, that do not apply to all the entities in the higher-level entity set. Indeed, the reason a designer applies specialization is to represent such distinctive features. If *student* and *employee* have exactly the same attributes as *person* entities, and participate in exactly the same relationships as *person* entities, there would be no need to specialize the *person* entity set.

Generalization proceeds from the recognition that a number of entity sets share some common features (namely, they are described by the same attributes and participate in the same relationship sets). On the basis of their commonalities, generalization synthesizes these entity sets into a single, higher-level entity set. Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences; it also permits an economy of representation in that shared attributes are not repeated.

6.8.3 Attribute Inheritance

A crucial property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**. The attributes of the higher-level entity sets are said to be **inherited** by the lower-level entity sets. For example, *student* and *employee* inherit the attributes of *person*. Thus, *student* is described by its *ID*, *name*, *street*, and *city* attributes, and additionally a *tot_cred* attribute; *employee* is described by its *ID*, *name*, *street*, and *city* attributes, and additionally a *salary* attribute. Attribute inheritance applies through all tiers of lower-level entity sets; thus, *instructor* and *secretary*, which are subclasses of *employee*, inherit the attributes *ID*, *name*, *street,* and *city* from *person*, in addition to inheriting *salary* from *employee*.

A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or superclass) participates. Like attribute inheritance, participation inheritance applies through all tiers of lower-level entity sets. For example, suppose the *person* entity set participates in a relationship *person_dept* with *department*. Then, the *student*, *employee*, *instructor* and *secretary* entity sets, which are subclasses of the *person* entity set, also implicitly participate in the *person_dept* relationship with *department*. These entity sets can participate in any relationships in which the *person* entity set participates.

Telegram: @uni_k

www.konkur.in

Whether a given portion of an E-R model was arrived at by specialization or generalization, the outcome is basically the same:

- A higher-level entity set with attributes and relationships that apply to all of its lower-level entity sets.
- Lower-level entity sets with distinctive features that apply only within a particular lower-level entity set.

In what follows, although we often refer to only generalization, the properties that we discuss belong fully to both processes.

Figure 6.18 depicts a **hierarchy** of entity sets. In the figure, *employee* is a lower-level entity set of *person* and a higher-level entity set of the *instructor* and *secretary* entity sets. In a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship; that is, entity sets in this diagram have only **single inheritance**. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a *lattice*.

6.8.4 Constraints on Specializations

To model an enterprise more accurately, the database designer may choose to place certain constraints on a particular generalization/specialization.

One type of constraint on specialization which we saw earlier specifies whether a specialization is disjoint or overlapping. Another type of constraint on a specialization/generalization is a **completeness constraint**, which specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

- **Total specialization** or **generalization**. Each higher-level entity must belong to a lower-level entity set.
- **Partial specialization** or **generalization**. Some higher-level entities may not belong to any lower-level entity set.

Partial specialization is the default. We can specify total specialization in an E-R diagram by adding the keyword "total" in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrowhead to which it applies (for a total specialization), or to the set of hollow arrowheads to which it applies (for an overlapping specialization).

The specialization of *person* to *student* or *employee* is total if the university does not need to represent any person who is neither a *student* nor an *employee*. However, if the university needs to represent such persons, then the specialization would be partial.

The completeness and disjointness constraints, do not depend on each other. Thus, specializations may be partial-overlapping, partial-disjoint, total-overlapping, and total-disjoint.

Telegram: @uni_k

www.konkur.in

We can see that certain insertion and deletion requirements follow from the constraints that apply to a given generalization or specialization. For instance, when a total completeness constraint is in place, an entity inserted into a higher-level entity set must also be inserted into at least one of the lower-level entity sets. An entity that is deleted from a higher-level entity set must also be deleted from all the associated lower-level entity sets to which it belongs.

6.8.5 Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship *proj_guide*, which we saw earlier, between an *instructor*, *student* and *project* (see Figure 6.6).

Now suppose that each instructor guiding a student on a project is required to file a monthly evaluation report. We model the evaluation report as an entity *evaluation*, with a primary key *evaluation_id*. One alternative for recording the (*student*, *project*, *instructor*) combination to which an *evaluation* corresponds is to create a quaternary (4-way) relationship set *eval_for* between *instructor*, *student*, *project*, and *evaluation*. (A quaternary relationship is required—a binary relationship between *student* and *evaluation*, for example, would not permit us to represent the (*project*, *instructor*) combination to which an *evaluation* corresponds.) Using the basic E-R modeling constructs, we obtain the E-R diagram of Figure 6.19. (We have omitted the attributes of the entity sets, for simplicity.)

It appears that the relationship sets *proj_guide* and *eval_for* can be combined into one single relationship set. Nevertheless, we should not combine them into a single

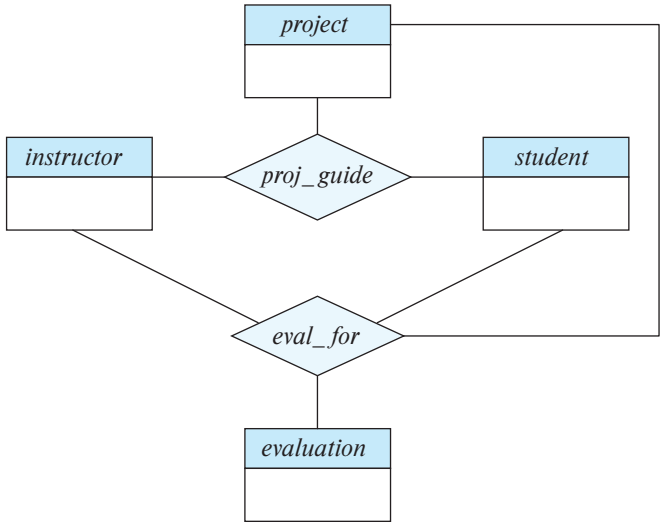


Figure 6.19 E-R diagram with redundant relationships.

Telegram: @uni_k

www.konkur.in

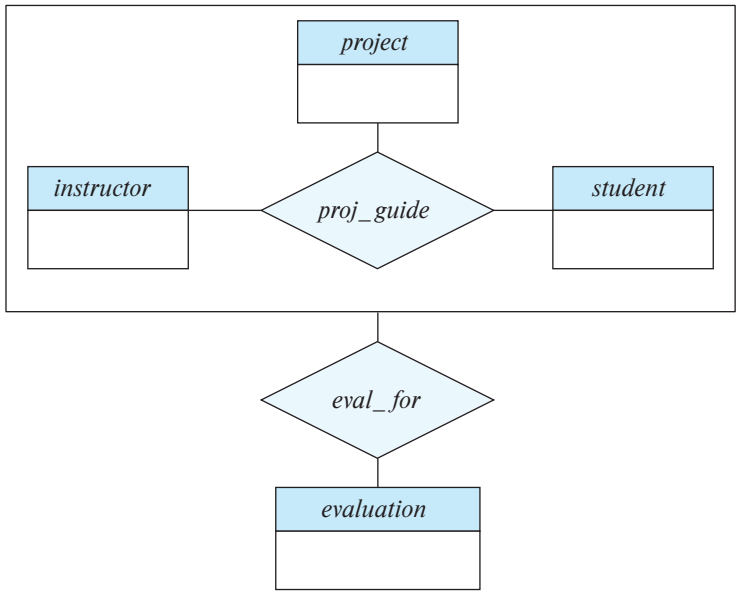


Figure 6.20 E-R diagram with aggregation.

relationship, since some *instructor*, *student*, *project* combinations may not have an associated *evaluation*.

There is redundant information in the resultant figure, however, since every *instructor*, *student*, *project* combination in *eval_for* must also be in *proj_guide*. If *evaluation* was modeled as a value rather than an entity, we could instead make *evaluation* a multivalued composite attribute of the relationship set *proj_guide*. However, this alternative may not be an option if an *evaluation* may also be related to other entities; for example, each evaluation report may be associated with a *secretary* who is responsible for further processing of the evaluation report to make scholarship payments.

The best way to model a situation such as the one just described is to use aggregation. **Aggregation** is an abstraction through which relationships are treated as higher-level entities. Thus, for our example, we regard the relationship set *proj_guide* (relating the entity sets *instructor*, *student*, and *project*) as a higher-level entity set called *proj_guide*. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship *eval_for* between *proj_guide* and *evaluation* to represent which (*student*, *project*, *instructor*) combination an *evaluation* is for. Figure 6.20 shows a notation for aggregation commonly used to represent this situation.

6.8.6 Reduction to Relation Schemas

We are in a position now to describe how the extended E-R features can be translated into relation schemas.

Telegram: @uni_k

www.konkur.in

6.8.6.1 Representation of Generalization

There are two different methods of designing relation schemas for an E-R diagram that includes generalization. Although we refer to the generalization in Figure 6.18 in this discussion, we simplify it by including only the first tier of lower-level entity sets—that is, *employee* and *student*. We assume that *ID* is the primary key of *person*.

1. Create a schema for the higher-level entity set. For each lower-level entity set, create a schema that includes an attribute for each of the attributes of that entity set plus one for each attribute of the primary key of the higher-level entity set. Thus, for the E-R diagram of Figure 6.18 (ignoring the *instructor* and *secretary* entity sets) we have three schemas:

   person (*ID*, *name, street, city*)
   employee (*ID*, *salary*)
   student (*ID*, *tot_cred*)

   The primary-key attributes of the higher-level entity set become primary-key attributes of the higher-level entity set as well as all lower-level entity sets. These can be seen underlined in the preceding example.

   In addition, we create foreign-key constraints on the lower-level entity sets, with their primary-key attributes referencing the primary key of the relation created from the higher-level entity set. In the preceding example, the *ID* attribute of *employee* would reference the primary key of *person*, and similarly for *student*.

2. An alternative representation is possible, if the generalization is disjoint and complete—that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher-level entity set is also a member of one of the lower-level entity sets. Here, we do not create a schema for the higher-level entity set. Instead, for each lower-level entity set, we create a schema that includes an attribute for each of the attributes of that entity set plus one for *each* attribute of the higher-level entity set. Then, for the E-R diagram of Figure 6.18, we have two schemas:

   employee (*ID*, *name*, *street*, *city*, *salary*)
   student (*ID*, *name*, *street*, *city*, *tot_cred*)

   Both these schemas have *ID*, which is the primary-key attribute of the higher-level entity set *person*, as their primary key.

One drawback of the second method lies in defining foreign-key constraints. To illustrate the problem, suppose we have a relationship set *R* involving entity set *person*. With the first method, when we create a relation schema *R* from the relationship set, we also define a foreign-key constraint on *R*, referencing the schema *person*. Unfortunately, with the second method, we do not have a single relation to which a foreign-key

Telegram: @uni_k

www.konkur.in

constraint on *R* can refer. To avoid this problem, we need to create a relation schema *person* containing at least the primary-key attributes of the *person* entity.

If the second method were used for an overlapping generalization, some values would be stored multiple times, unnecessarily. For instance, if a person is both an employee and a student, values for *street* and *city* would be stored twice.

If the generalization were disjoint but not complete—that is, if some person is neither an employee nor a student—then an extra schema

person (*ID*, *name, street, city*)

would be required to represent such people. However, the problem with foreign-key constraints mentioned above would remain. As an attempt to work around the problem, suppose employees and students are additionally represented in the *person* relation. Unfortunately, name, street, and city information would then be stored redundantly in the *person* relation and the *student* relation for students, and similarly in the *person* relation and the *employee* relation for employees. That suggests storing name, street, and city information only in the *person* relation and removing that information from *student* and *employee*. If we do that, the result is exactly the first method we presented.

6.8.6.2 Representation of Aggregation

Designing schemas for an E-R diagram containing aggregation is straightforward. Consider Figure 6.20. The schema for the relationship set *eval_for* between the aggregation of *proj_guide* and the entity set *evaluation* includes an attribute for each attribute in the primary keys of the entity set *evaluation* and the relationship set *proj_guide*. It also includes an attribute for any descriptive attributes, if they exist, of the relationship set *eval_for*. We then transform the relationship sets and entity sets within the aggregated entity set following the rules we have already defined.

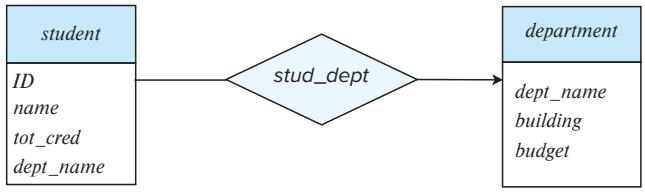
The rules we saw earlier for creating primary-key and foreign-key constraints on relationship sets can be applied to relationship sets involving aggregations as well, with the aggregation treated like any other entity set. The primary key of the aggregation is the primary key of its defining relationship set. No separate relation is required to represent the aggregation; the relation created from the defining relationship is used instead.

6.9 Entity-Relationship Design Issues

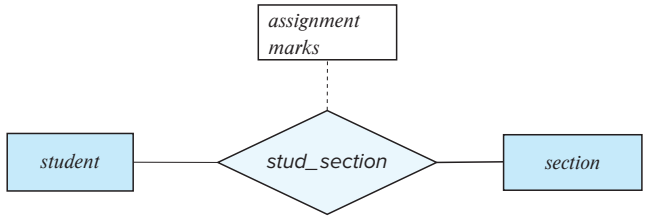
The notions of an entity set and a relationship set are not precise, and it is possible to define a set of entities and the relationships among them in a number of different ways. In this section, we examine basic issues in the design of an E-R database schema. Section 6.11 covers the design process in further detail.

Telegram: @uni_k

www.konkur.in



(a) Incorrect use of attribute



(b) Erroneous use of relationship attributes

Figure 6.21 Example of erroneous E-R diagrams

6.9.1 Common Mistakes in E-R Diagrams

A common mistake when creating E-R models is the use of the primary key of an entity set as an attribute of another entity set, instead of using a relationship. For example, in our university E-R model, it is incorrect to have *dept_name* as an attribute of *student*, as depicted in Figure 6.21a, even though it is present as an attribute in the relation schema for *student*. The relationship *stud_dept* is the correct way to represent this information in the E-R model, since it makes the relationship between *student* and *department* explicit, rather than implicit via an attribute. Having an attribute *dept_name* as well as a relationship *stud_dept* would result in duplication of information.

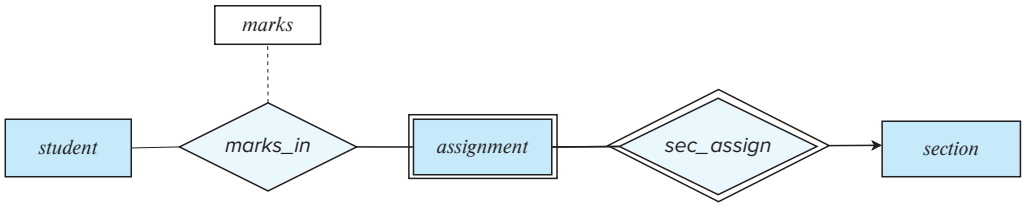
Another related mistake that people sometimes make is to designate the primary-key attributes of the related entity sets as attributes of the relationship set. For example, *ID* (the primary-key attributes of *student*) and *ID* (the primary key of *instructor*) should not appear as attributes of the relationship *advisor*. This should not be done since the primary-key attributes are already implicit in the relationship set.[6]

A third common mistake is to use a relationship with a single-valued attribute in a situation that requires a multivalued attribute. For example, suppose we decided to represent the marks that a student gets in different assignments of a course offering (*section*). A wrong way of doing this would be to add two attributes *assignment* and *marks* to the relationship *takes*, as depicted in Figure 6.21b. The problem with this design is that we can only represent a single assignment for a given student-section pair,

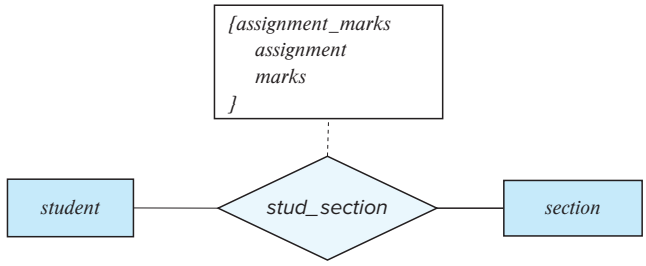
[6]When we create a relation schema from the E-R schema, the attributes may appear in a schema created from the *advisor* relationship set, as we shall see later; however, they should not appear in the

Telegram: @uni_k

www.konkur.in



(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Figure 6.22 Correct versions of the E-R diagram of Figure 6.21.

since relationship instances must be uniquely identified by the participating entities, *student* and *section*.

One solution to the problem depicted in Figure 6.21c, shown in Figure 6.22a, is to model *assignment* as a weak entity identified by *section*, and to add a relationship *marks_in* between *assignment* and *student*; the relationship would have an attribute *marks*. An alternative solution, shown in Figure 6.22d, is to use a multivalued composite attribute {*assignment_marks*} to *takes*, where *assignment_marks* has component attributes *assignment* and *marks*. Modeling an assignment as a weak entity is preferable in this case, since it allows recording other information about the assignment, such as maximum marks or deadlines.

When an E-R diagram becomes too big to draw in a single piece, it makes sense to break it up into pieces, each showing part of the E-R model. When doing so, you may need to depict an entity set in more than one page. As discussed in Section 6.2.2, attributes of the entity set should be shown only once, in its first occurrence. Subsequent occurrences of the entity set should be shown without any attributes, to avoid repeating the same information at multiple places, which may lead to inconsistency.

6.9.2 Use of Entity Sets versus Attributes

Consider the entity set *instructor* with the additional attribute *phone_number* (Figure 6.23a.) It can be argued that a phone is an entity in its own right with attributes *phone*

Telegram: @uni_k

www.konkur.in

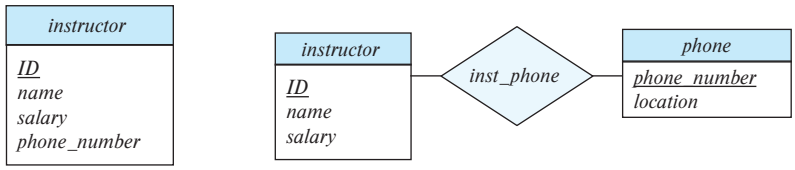


Figure 6.23 Alternatives for adding *phone* to the *instructor* entity set.

_number and *location*; the location may be the office or home where the phone is located, with mobile (cell) phones perhaps represented by the value "mobile." If we take this point of view, we do not add the attribute *phone_number* to the *instructor*. Rather, we create:

- A *phone* entity set with attributes *phone_number* and *location*.
- A relationship set *inst_phone*, denoting the association between instructors and the phones that they have.

This alternative is shown in Figure 6.23b.

What, then, is the main difference between these two definitions of an instructor? Treating a phone as an attribute *phone_number* implies that instructors have precisely one phone number each. Treating a phone as an entity *phone* permits instructors to have several phone numbers (including zero) associated with them. However, we could instead easily define *phone_number* as a multivalued attribute to allow multiple phones per instructor.

The main difference then is that treating a phone as an entity better models a situation where one may want to keep extra information about a phone, such as its location, or its type (mobile, IP phone, or plain old phone), or all who share the phone. Thus, treating phone as an entity is more general than treating it as an attribute and is appropriate when the generality may be useful.

In contrast, it would not be appropriate to treat the attribute *name* (of an instructor) as an entity; it is difficult to argue that *name* is an entity in its own right (in contrast to the phone). Thus, it is appropriate to have *name* as an attribute of the *instructor* entity set.

Two natural questions thus arise: What constitutes an attribute, and what constitutes an entity set? Unfortunately, there are no simple answers. The distinctions mainly depend on the structure of the real-world enterprise being modeled and on the semantics associated with the attribute in question.

6.9.3 Use of Entity Sets versus Relationship Sets

It is not always clear whether an object is best expressed by an entity set or a relationship set. In Figure 6.15, we used the *takes* relationship set to model the situation where a

Telegram: @uni_k

www.konkur.in

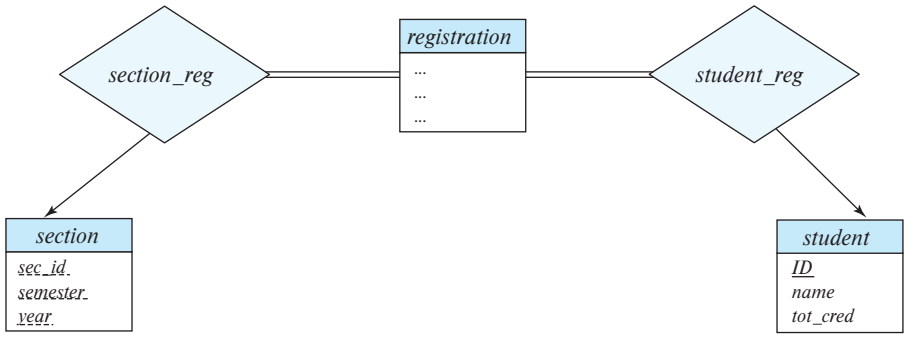


Figure 6.24 Replacement of *takes* by *registration* and two relationship sets.

student takes a (section of a) course. An alternative is to imagine that there is a course-registration record for each course that each student takes. Then, we have an entity set to represent the course-registration record. Let us call that entity set *registration*. Each *registration* entity is related to exactly one student and to exactly one section, so we have two relationship sets, one to relate course-registration records to students and one to relate course-registration records to sections. In Figure 6.24, we show the entity sets *section* and *student* from Figure 6.15 with the *takes* relationship set replaced by one entity set and two relationship sets:

- *registration*, the entity set representing course-registration records.
- *section_reg*, the relationship set relating *registration* and *course*.
- *student_reg*, the relationship set relating *registration* and *student*.

Note that we use double lines to indicate total participation by *registration* entities.

Both the approach of Figure 6.15 and that of Figure 6.24 accurately represent the university's information, but the use of *takes* is more compact and probably preferable. However, if the registrar's office associates other information with a course-registration record, it might be best to make it an entity in its own right.

One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

6.9.4 Binary versus *n*-ary Relationship Sets

Relationships in databases are often binary. Some relationships that appear to be nonbinary could actually be better represented by several binary relationships. For instance, one could create a ternary relationship *parent*, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, *mother* and *father*, relating a child to his/her mother and father separately. Using

Telegram: @uni_k

www.konkur.in

the two relationships *mother* and *father* provides us with a record of a child's mother, even if we are not aware of the father's identity; a null value would be required if the ternary relationship *parent* were used. Using binary relationship sets is preferable in this case.

In fact, it is always possible to replace a nonbinary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary ($n = 3$) relationship set R, relating entity sets A, B, and C. We replace the relationship set R with an entity set E, and we create three relationship sets as shown in Figure 6.25:

- R_A, a many-to-one relationship set from E to A.
- R_B, a many-to-one relationship set from E to B.
- R_C, a many-to-one relationship set from E to C.

E is required to have total participation in each of R_A, R_B, and R_C. If the relationship set R had any attributes, these are assigned to entity set E; further, a special identifying attribute is created for E (since it must be possible to distinguish different entities in an entity set on the basis of their attribute values). For each relationship (a_i, b_i, c_i) in the relationship set R, we create a new entity e_i in the entity set E. Then, in each of the three new relationship sets, we insert a relationship as follows:

- (e_i, a_i) in R_A.
- (e_i, b_i) in R_B.
- (e_i, c_i) in R_C.

We can generalize this process in a straightforward manner to n-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

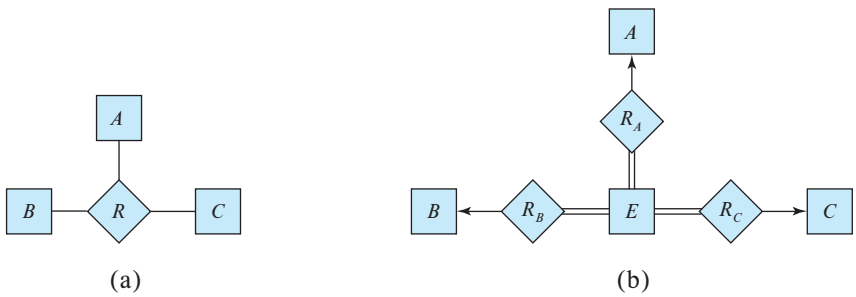


Figure 6.25 Ternary relationship versus three binary relationships.

Telegram: @uni_k

www.konkur.in

- An identifying attribute may have to be created for the entity set created to represent the relationship set. This attribute, along with the extra relationship sets required, increases the complexity of the design and (as we shall see in Section 6.7) overall storage requirements.
- An n-ary relationship set shows more clearly that several entities participate in a single relationship.
- There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships. For example, consider a constraint that says that R is many-to-one from A, B to C; that is, each pair of entities from A and B is associated with at most one C entity. This constraint cannot be expressed by using cardinality constraints on the relationship sets R_A, R_B, and R_C.

Consider the relationship set *proj_guide* in Section 6.2.2, relating *instructor*, *student*, and *project*. We cannot directly split *proj_guide* into binary relationships between *instructor* and *project* and between *instructor* and *student*. If we did so, we would be able to record that instructor Katz works on projects A and B with students Shankar and Zhang; however, we would not be able to record that Katz works on project A with student Shankar and works on project B with student Zhang, but does not work on project A with Zhang or on project B with Shankar.

The relationship set *proj_guide* can be split into binary relationships by creating a new entity set as described above. However, doing so would not be very natural.

6.10 Alternative Notations for Modeling Data

A diagrammatic representation of the data model of an application is a very important part of designing a database schema. Creation of a database schema requires not only data modeling experts, but also domain experts who know the requirements of the application but may not be familiar with data modeling. An intuitive diagrammatic representation is particularly important since it eases communication of information between these groups of experts.

A number of alternative notations for modeling data have been proposed, of which E-R diagrams and UML class diagrams are the most widely used. There is no universal standard for E-R diagram notation, and different books and E-R diagram software use different notations.

In the rest of this section, we study some of the alternative E-R diagram notations, as well as the UML class diagram notation. To aid in comparison of our notation with these alternatives, Figure 6.26 summarizes the set of symbols we have used in our E-R diagram notation.

6.10.1 Alternative E-R Notations

Figure 6.27 indicates some of the alternative E-R notations that are widely used. One alternative representation of attributes of entities is to show them in ovals connected

Telegram: @uni_k

www.konkur.in

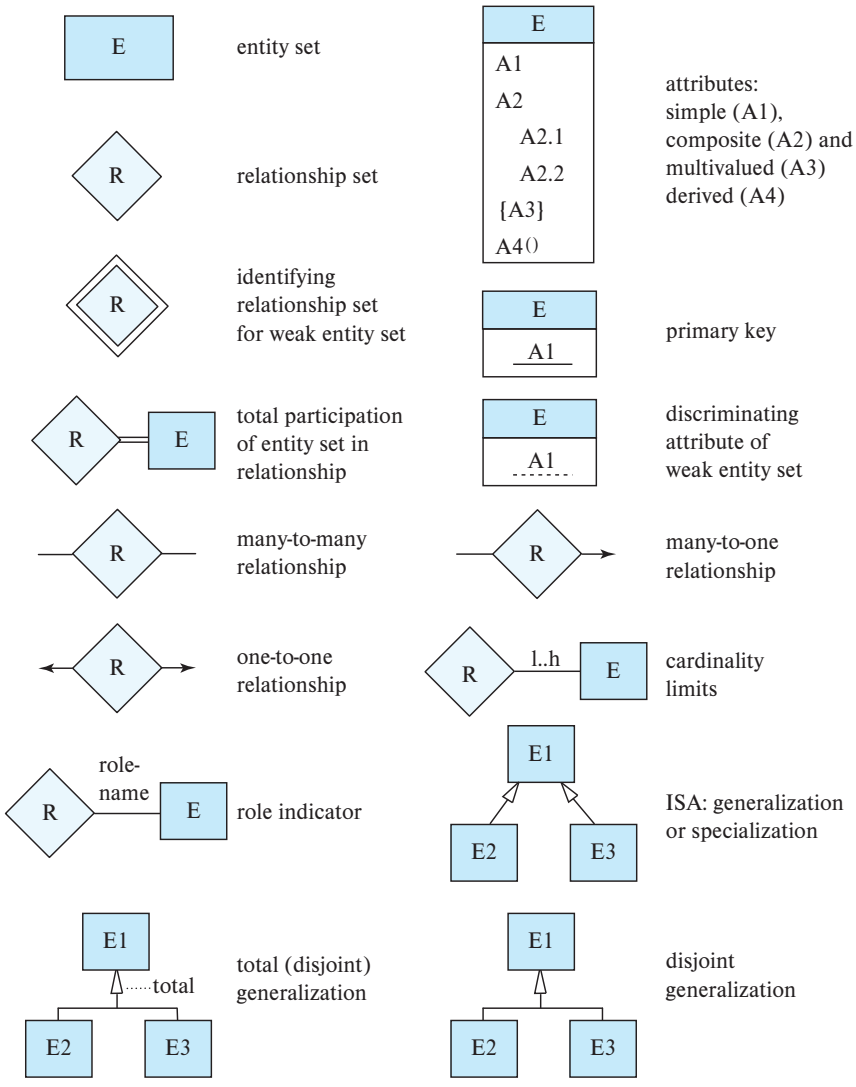


Figure 6.26 Symbols used in the E-R notation.

to the box representing the entity; primary key attributes are indicated by underlining them. The above notation is shown at the top of the figure. Relationship attributes can be similarly represented, by connecting the ovals to the diamond representing the relationship.

Cardinality constraints on relationships can be indicated in several different ways, as shown in Figure 6.27. In one alternative, shown on the left side of the figure, labels $*$ and 1 on the edges out of the relationship are used for depicting many-to-many, one-

Telegram: @uni_k

www.konkur.in

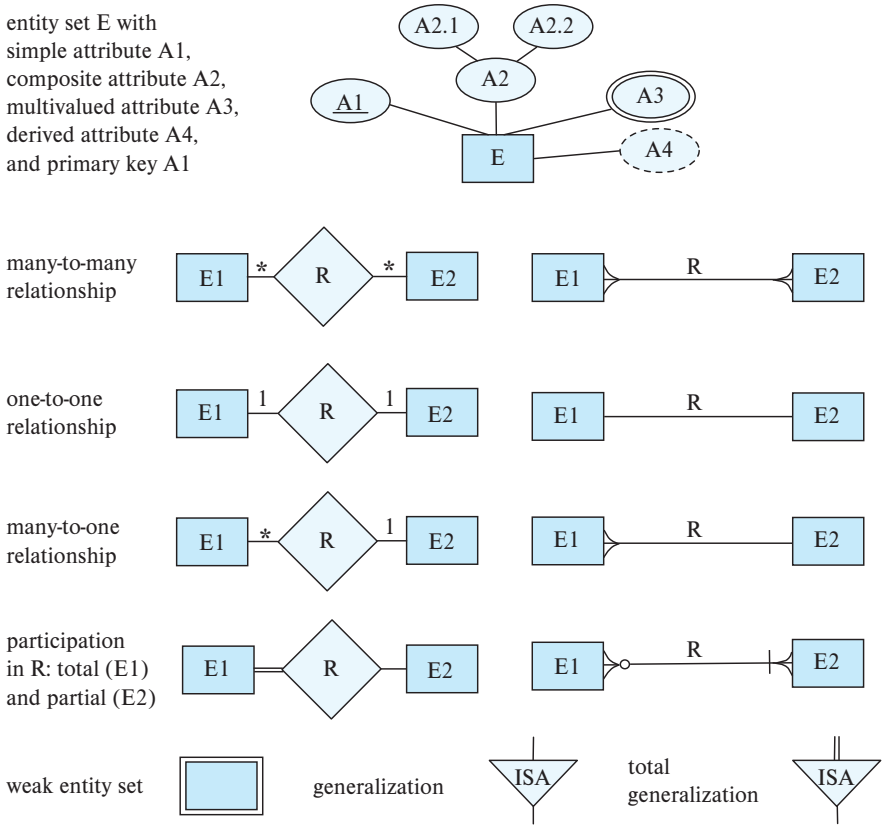


Figure 6.27 Alternative E-R notations.

to-one, and many-to-one relationships. The case of one-to-many is symmetric to many-to-one and is not shown.

In another alternative notation shown on the right side of Figure 6.27, relationship sets are represented by lines between entity sets, without diamonds; only binary relationships can be modeled thus. Cardinality constraints in such a notation are shown by "crow's-foot" notation, as in the figure. In a relationship R between $E1$ and $E2$, crow's feet on both sides indicate a many-to-many relationship, while crow's feet on just the $E1$ side indicate a many-to-one relationship from $E1$ to $E2$. Total participation is specified in this notation by a vertical bar. Note however, that in a relationship R between entities $E1$ and $E2$, if the participation of $E1$ in R is total, the vertical bar is placed on the opposite side, adjacent to entity $E2$. Similarly, partial participation is indicated by using a circle, again on the opposite side.

The bottom part of Figure 6.27 shows an alternative representation of generalization, using triangles instead of hollow arrowheads.

Telegram: @uni_k

www.konkur.in

In prior editions of this text up to the fifth edition, we used ovals to represent attributes, with triangles representing generalization, as shown in Figure 6.27. The notation using ovals for attributes and diamonds for relationships is close to the original form of E-R diagrams used by Chen in his paper that introduced the notion of E-R modeling. That notation is now referred to as Chen's notation.

The U.S. National Institute for Standards and Technology defined a standard called IDEF1X in 1993. IDEF1X uses the crow's-foot notation, with vertical bars on the relationship edge to denote total participation and hollow circles to denote partial participation, and it includes other notations that we have not shown.

With the growth in the use of Unified Markup Language (UML), described in Section 6.10.2, we have chosen to update our E-R notation to make it closer to the form of UML class diagrams; the connections will become clear in Section 6.10.2. In comparison with our previous notation, our new notation provides a more compact representation of attributes, and it is also closer to the notation supported by many E-R modeling tools, in addition to being closer to the UML class diagram notation.

There are a variety of tools for constructing E-R diagrams, each of which has its own notational variants. Some of the tools even provide a choice between several E-R notation variants. See the tools section at the end of the chapter for references.

One key difference between entity sets in an E-R diagram and the relation schemas created from such entities is that attributes in the relational schema corresponding to E-R relationships, such as the *dept_name* attribute of *instructor*, are not shown in the entity set in the E-R diagram. Some data modeling tools allow designers to choose between two views of the same entity, one an entity view without such attributes, and other a relational view with such attributes.

6.10.2 The Unified Modeling Language UML

Entity-relationship diagrams help model the data representation component of a software system. Data representation, however, forms only one part of an overall system design. Other components include models of user interactions with the system, specification of functional modules of the system and their interaction, etc. The **Unified Modeling Language** (**UML**) is a standard developed under the auspices of the **Object Management Group** (**OMG**) for creating specifications of various components of a software system. Some of the parts of UML are:

- **Class diagram**. A class diagram is similar to an E-R diagram. Later in this section we illustrate a few features of class diagrams and how they relate to E-R diagrams.
- **Use case diagram**. Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).
- **Activity diagram**. Activity diagrams depict the flow of tasks between various components of a system.

Telegram: @uni_k

www.konkur.in

- **Implementation diagram**. Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

We do not attempt to provide detailed coverage of the different parts of UML here. Instead we illustrate some features of that part of UML that relates to data modeling through examples. See the Further Reading section at the end of the chapter for references on UML.

Figure 6.28 shows several E-R diagram constructs and their equivalent UML class diagram constructs. We describe these constructs below. UML actually models objects, whereas E-R models entities. Objects are like entities, and have attributes, but additionally provide a set of functions (called methods) that can be invoked to compute values on the basis of attributes of the objects, or to update the object itself. Class diagrams can depict methods in addition to attributes. We cover objects in Section 8.2. UML does not support composite or multivalued attributes, and derived attributes are equivalent to methods that take no parameters. Since classes support encapsulation, UML allows attributes and methods to be prefixed with a "+", "-", or "#", which denote respectively public, private, and protected access. Private attributes can only be used in methods of the class, while protected attributes can be used only in methods of the class and its subclasses; these should be familiar to anyone who knows Java, C++, or C#.

In UML terminology, relationship sets are referred to as **associations**; we shall refer to them as relationship sets for consistency with E-R terminology. We represent binary relationship sets in UML by just drawing a line connecting the entity sets. We write the relationship set name adjacent to the line. We may also specify the role played by an entity set in a relationship set by writing the role name on the line, adjacent to the entity set. Alternatively, we may write the relationship set name in a box, along with attributes of the relationship set, and connect the box by a dotted line to the line depicting the relationship set. This box can then be treated as an entity set, in the same way as an aggregation in E-R diagrams, and can participate in relationships with other entity sets.

Since UML version 1.3, UML supports nonbinary relationships, using the same diamond notation used in E-R diagrams. Nonbinary relationships could not be directly represented in earlier versions of UML—they had to be converted to binary relationships by the technique we have seen earlier in Section 6.9.4. UML allows the diamond notation to be used even for binary relationships, but most designers use the line notation.

Cardinality constraints are specified in UML in the same way as in E-R diagrams, in the form $l..h$, where l denotes the minimum and h the maximum number of relationships an entity can participate in. However, you should be aware that the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams, as shown in Figure 6.28. The constraint $0..*$ on the $E2$ side and $0..1$ on the $E1$ side means that each $E2$ entity can participate in at most one relationship, whereas each $E1$ entity can participate in many relationships; in other words, the relationship is many-to-one from $E2$ to $E1$.

Telegram: @uni_k

www.konkur.in

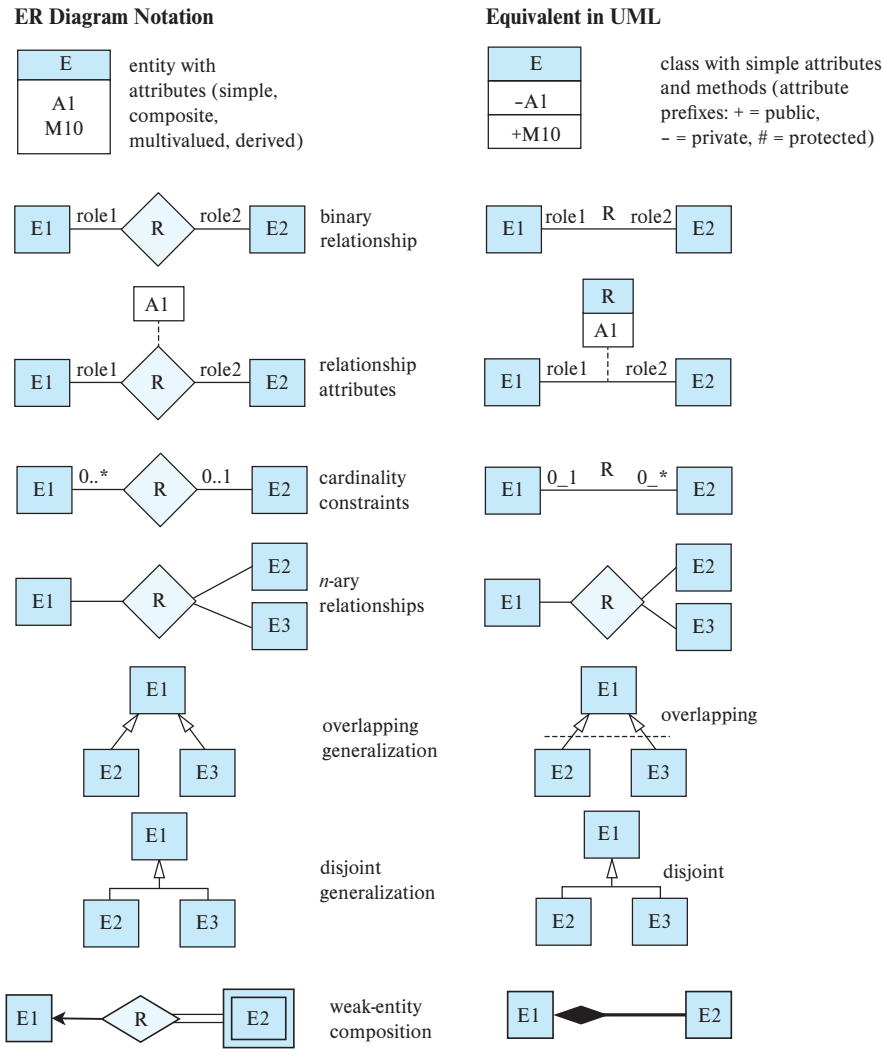


Figure 6.28 Symbols used in the UML class diagram notation.

Single values such as 1 or $*$ may be written on edges; the single value 1 on an edge is treated as equivalent to 1..1, while $*$ is equivalent to 0.. $*$. UML supports generalization; the notation is basically the same as in our E-R notation, including the representation of disjoint and overlapping generalizations.

UML class diagrams include several other notations that approximately correspond to the E-R notations we have seen. A line between two entity sets with a small shaded diamond at one end in UML specifies "composition" in UML. The composition relationship between $E2$ and $E1$ in Figure 6.28 indicates that $E2$ is existence dependent on $E1$; this is roughly equivalent to denoting $E2$ as a weak entity set that is existence

Telegram: @uni_k

www.konkur.in

dependent on the identifying entity set $E1$. (The term *aggregation* in UML denotes a variant of composition where $E2$ is contained in $E1$ but may exist independently, and it is denoted using a small hollow diamond.)

UML class diagrams also provide notations to represent object-oriented language features such as interfaces. See the Further Reading section for more information on UML class diagrams.

6.11 Other Aspects of Database Design

Our extensive discussion of schema design in this chapter may create the false impression that schema design is the only component of a database design. There are indeed several other considerations that we address more fully in subsequent chapters, and survey briefly here.

6.11.1 Functional Requirements

All enterprises have rules on what kinds of functionality are to be supported by an enterprise application. These could include transactions that update the data, as well as queries to view data in a desired fashion. In addition to planning the functionality, designers have to plan the interfaces to be built to support the functionality.

Not all users are authorized to view all data, or to perform all transactions. An authorization mechanism is very important for any enterprise application. Such authorization could be at the level of the database, using database authorization features. But it could also be at the level of higher-level functionality or interfaces, specifying who can use which functions/interfaces.

6.11.2 Data Flow, Workflow

Database applications are often part of a larger enterprise application that interacts not only with the database system but also with various specialized applications. As an example, consider a travel-expense report. It is created by an employee returning from a business trip (possibly by means of a special software package) and is subsequently routed to the employee's manager, perhaps other higher-level managers, and eventually to the accounting department for payment (at which point it interacts with the enterprise's accounting information systems).

The term *workflow* refers to the combination of data and tasks involved in processes like those of the preceding examples. Workflows interact with the database system as they move among users and users perform their tasks on the workflow. In addition to the data on which workflows operate, the database may store data about the workflow itself, including the tasks making up a workflow and how they are to be routed among users. Workflows thus specify a series of queries and updates to the database that may be taken into account as part of the database-design process. Put in other terms, modeling the

Telegram: @uni_k

www.konkur.in

enterprise requires us not only to understand the semantics of the data but also the business processes that use those data.

6.11.3 Schema Evolution

Database design is usually not a one-time activity. The needs of an organization evolve continually, and the data that it needs to store also evolve correspondingly. During the initial database-design phases, or during the development of an application, the database designer may realize that changes are required at the conceptual, logical, or physical schema levels. Changes in the schema can affect all aspects of the database application. A good database design anticipates future needs of an organization and ensures that the schema requires minimal changes as the needs evolve.

It is important to distinguish between fundamental constraints that are expected to be permanent and constraints that are anticipated to change. For example, the constraint that an instructor-id identify a unique instructor is fundamental. On the other hand, a university may have a policy that an instructor can have only one department, which may change at a later date if joint appointments are allowed. A database design that only allows one department per instructor might require major changes if joint appointments are allowed. Such joint appointments can be represented by adding an extra relationship without modifying the *instructor* relation, as long as each instructor has only one primary department affiliation; a policy change that allows more than one primary affiliation may require a larger change in the database design. A good design should account not only for current policies, but should also avoid or minimize the need for modifications due to changes that are anticipated or have a reasonable chance of happening.

Finally, it is worth noting that database design is a human-oriented activity in two senses: the end users of the system are people (even if an application sits between the database and the end users); and the database designer needs to interact extensively with experts in the application domain to understand the data requirements of the application. All of the people involved with the data have needs and preferences that should be taken into account in order for a database design and deployment to succeed within the enterprise.

6.12 Summary

- Database design mainly involves the design of the database schema. The entity-relationship (E-R) data model is a widely used data model for database design. It provides a convenient graphical representation to view data, relationships, and constraints.
- The E-R model is intended primarily for the database-design process. It was developed to facilitate database design by allowing the specification of an enterprise schema. Such a schema represents the overall logical structure of the database. This overall structure can be expressed graphically by an diagram.

Telegram: @uni_k

www.konkur.in

- An entity is an object that exists in the real world and is distinguishable from other objects. We express the distinction by associating with each entity a set of attributes that describes the object.
- A relationship is an association among several entities. A relationship set is a collection of relationships of the same type, and an entity set is a collection of entities of the same type.
- The terms superkey, candidate key, and primary key apply to entity and relationship sets as they do for relation schemas. Identifying the primary key of a relationship set requires some care, since it is composed of attributes from one or more of the related entity sets.
- Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.
- An entity set that does not have sufficient attributes to form a primary key is termed a weak entity set. An entity set that has a primary key is termed a strong entity set.
- The various features of the E-R model offer the database designer numerous choices in how to best represent the enterprise being modeled. Concepts and objects may, in certain cases, be represented by entities, relationships, or attributes. Aspects of the overall structure of the enterprise may be best described by using weak entity sets, generalization, specialization, or aggregation. Often, the designer must weigh the merits of a simple, compact model versus those of a more precise, but more complex one.
- A database design specified by an E-R diagram can be represented by a collection of relation schemas. For each entity set and for each relationship set in the database, there is a unique relation schema that is assigned the name of the corresponding entity set or relationship set. This forms the basis for deriving a relational database design from an E-R diagram.
- Specialization and generalization define a containment relationship between a higher-level entity set and one or more lower-level entity sets. Specialization is the result of taking a subset of a higher-level entity set to form a lower-level entity set. Generalization is the result of taking the union of two or more disjoint (lower-level) entity sets to produce a higher-level entity set. The attributes of higher-level entity sets are inherited by lower-level entity sets.
- Aggregation is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships.
- Care must be taken in E-R design. There are a number of common mistakes to avoid. Also, there are choices among the use of entity sets, relationship sets, and

Telegram: @uni_k

www.konkur.in

attributes in representing aspects of the enterprise whose correctness may depend on subtle details specific to the enterprise.

- UML is a popular modeling language. UML class diagrams are widely used for modeling classes, as well as for general-purpose data modeling.

Review Terms

- Design Process
  - Conceptual-design
  - Logical-design
  - Physical-design
- Entity-relationship (E-R) data model
- Entity and entity set
  - Simple and composite attributes
  - Single-valued and multivalued attributes
  - Derived attribute
- Key
  - Superkey
  - Candidate key
  - Primary key
- Relationship and relationship set
  - Binary relationship set
  - Degree of relationship set
  - Descriptive attributes
  - Superkey, candidate key, and primary key
  - Role
  - Recursive relationship set
- E-R diagram
- Mapping cardinality:
  - One-to-one relationship
  - One-to-many relationship
  - Many-to-one relationship
  - Many-to-many relationship
- Total and partial participation
- Weak entity sets and strong entity sets
  - Discriminator attributes
  - Identifying relationship
- Specialization and generalization
- Aggregation
- Design choices
- United Modeling Language (UML)

Practice Exercises

6.1 Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

Telegram: @uni_k

www.konkur.in

6.2 Consider a database that includes the entity sets *student*, *course*, and *section* from the university schema and that additionally records the marks that students receive in different exams of different sections.

a. Construct an E-R diagram that models exams as entities and uses a ternary relationship as part of the design.

b. Construct an alternative E-R diagram that uses only a binary relationship between *student* and *section*. Make sure that only one relationship exists between a particular *student* and *section* pair, yet you can represent the marks that a student gets in different exams.

6.3 Design an E-R diagram for keeping track of the scoring statistics of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player scoring statistics for each match. Summary statistics should be modeled as derived attributes with an explanation as to how they are computed.

6.4 Consider an E-R diagram in which the same entity set appears several times, with its attributes repeated in more than one occurrence. Why is allowing this redundancy a bad practice that one should avoid?

6.5 An E-R diagram can be viewed as a graph. What do the following mean in terms of the structure of an enterprise schema?

a. The graph is disconnected.

b. The graph has a cycle.

6.6 Consider the representation of the ternary relationship of Figure 6.29a using the binary relationships illustrated in Figure 6.29b (attributes not shown).

a. Show a simple instance of E, A, B, C, R_A, R_B, and R_C that cannot correspond to any instance of A, B, C, and R.

b. Modify the E-R diagram of Figure 6.29b to introduce constraints that will guarantee that any instance of E, A, B, C, R_A, R_B, and R_C that satisfies the constraints will correspond to an instance of A, B, C, and R.

c. Modify the preceding translation to handle total participation constraints on the ternary relationship.

6.7 A weak entity set can always be made into a strong entity set by adding to its attributes the primary-key attributes of its identifying entity set. Outline what sort of redundancy will result if we do so.

6.8 Consider a relation such as *sec_course*, generated from a many-to-one relationship set *sec_course*. Do the primary and foreign key constraints created on the relation enforce the many-to-one cardinality constraint? Explain why.

Telegram: @uni_k

www.konkur.in

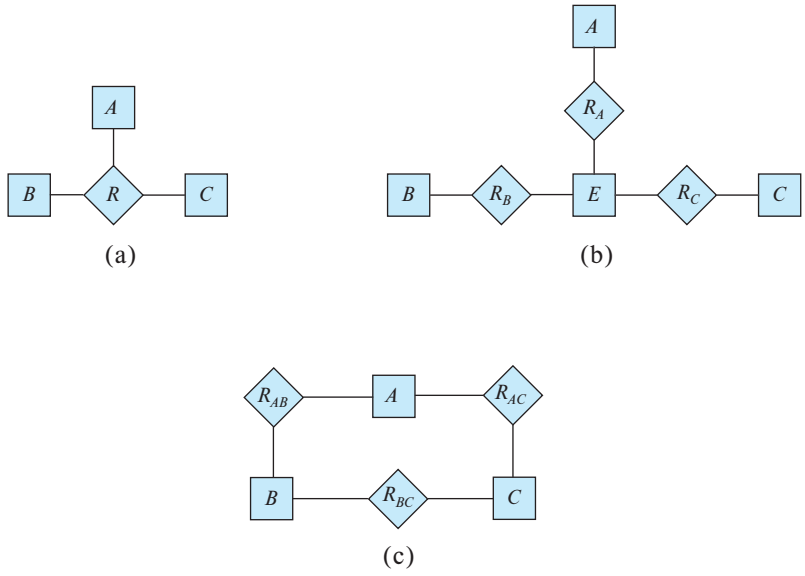


Figure 6.29 Representation of a ternary relationship using binary relationships.

6.9 Suppose the *advisor* relationship set were one-to-one. What extra constraints are required on the relation *advisor* to ensure that the one-to-one cardinality constraint is enforced?

6.10 Consider a many-to-one relationship R between entity sets A and B. Suppose the relation created from R is combined with the relation created from A. In SQL, attributes participating in a foreign key constraint can be null. Explain how a constraint on total participation of A in R can be enforced using **not null** constraints in SQL.

6.11 In SQL, foreign key constraints can reference only the primary key attributes of the referenced relation or other attributes declared to be a superkey using the **unique** constraint. As a result, total participation constraints on a many-to-many relationship set (or on the "one" side of a one-to-many relationship set) cannot be enforced on the relations created from the relationship set, using primary key, foreign key, and not null constraints on the relations.

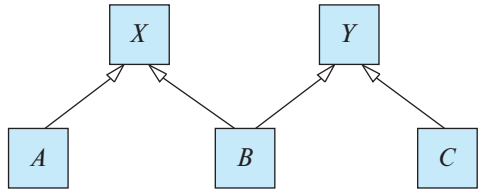
a. Explain why.

b. Explain how to enforce total participation constraints using complex check constraints or assertions (see Section 4.4.8). (Unfortunately, these features are not supported on any widely used database currently.)

6.12 Consider the following lattice structure of generalization and specialization (attributes not shown).

Telegram: @uni_k

www.konkur.in



For entity sets A, B, and C, explain how attributes are inherited from the higher-level entity sets X and Y. Discuss how to handle a case where an attribute of X has the same name as some attribute of Y.

6.13 An E-R diagram usually models the state of an enterprise at a point in time. Suppose we wish to track *temporal changes*, that is, changes to data over time. For example, Zhang may have been a student between September 2015 and May 2019, while Shankar may have had instructor Einstein as advisor from May 2018 to December 2018, and again from June 2019 to January 2020. Similarly, attribute values of an entity or relationship, such as *title* and *credits* of *course*, *salary*, or even *name* of *instructor*, and *tot_cred* of *student*, can change over time.

One way to model temporal changes is as follows: We define a new data type called **valid_time**, which is a time interval, or a set of time intervals. We then associate a *valid_time* attribute with each entity and relationship, recording the time periods during which the entity or relationship is valid. The end time of an interval can be infinity; for example, if Shankar became a student in September 2018, and is still a student, we can represent the end time of the *valid_time* interval as infinity for the Shankar entity. Similarly, we model attributes that can change over time as a set of values, each with its own *valid_time*.

a. Draw an E-R diagram with the *student* and *instructor* entities, and the *advisor* relationship, with the above extensions to track temporal changes.

b. Convert the E-R diagram discussed above into a set of relations.

It should be clear that the set of relations generated is rather complex, leading to difficulties in tasks such as writing queries in SQL. An alternative approach, which is used more widely, is to ignore temporal changes when designing the E-R model (in particular, temporal changes to attribute values), and to modify the relations generated from the E-R model to track temporal changes.

Exercises

6.14 Explain the distinctions among the terms *primary key*, *candidate key*, and *superkey*.

Telegram: @uni_k

www.konkur.in

6.15 Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

6.16 Extend the E-R diagram of Exercise 6.3 to track the same information for all teams in a league.

6.17 Explain the difference between a weak and a strong entity set.

6.18 Consider two entity sets A and B that both have the attribute X (among others whose names are not relevant to this question).

a. If the two Xs are completely unrelated, how should the design be improved?

b. If the two Xs represent the same property and it is one that applies both to A and to B, how should the design be improved? Consider three subcases:

- X is the primary key for A but not B
- X is the primary key for both A and B
- X is not the primary key for A nor for B

6.19 We can convert any weak entity set to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entity sets?

6.20 Construct appropriate relation schemas for each of the E-R diagrams in:

a. Exercise 6.1.

b. Exercise 6.2.

c. Exercise 6.3.

d. Exercise 6.15.

6.21 Consider the E-R diagram in Figure 6.30, which models an online bookstore.

a. Suppose the bookstore adds Blu-ray discs and downloadable video to its collection. The same item may be present in one or both formats, with differing prices. Draw the part of the E-R diagram that models this addition, showing just the parts related to video.

b. Now extend the full E-R diagram to model the case where a shopping basket may contain any combination of books, Blu-ray discs, or downloadable video.

6.22 Design a database for an automobile company to provide to its dealers to assist them in maintaining customer records and dealer inventory and to assist sales staff in ordering cars.

Telegram: @uni_k

www.konkur.in

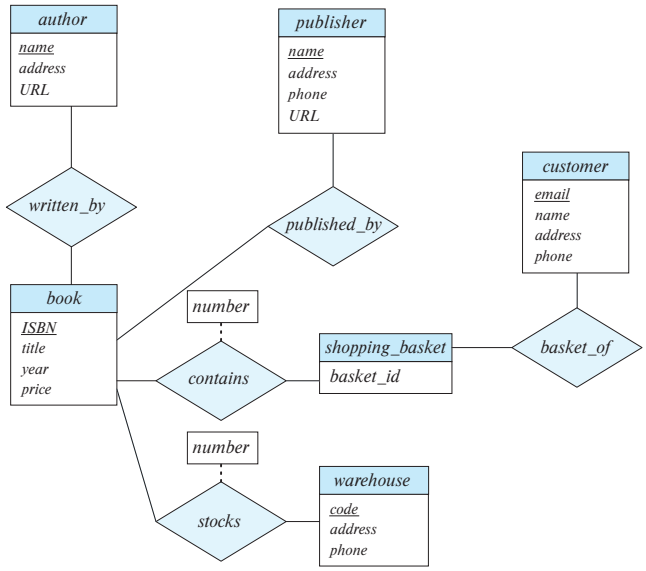


Figure 6.30 E-R diagram for modeling an online bookstore.

Each vehicle is identified by a vehicle identification number (VIN). Each individual vehicle is a particular model of a particular brand offered by the company (e.g., the XF is a model of the car brand Jaguar of Tata Motors). Each model can be offered with a variety of options, but an individual car may have only some (or none) of the available options. The database needs to store information about models, brands, and options, as well as information about individual dealers, customers, and cars.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

6.23 Design a database for a worldwide package delivery company (e.g., DHL or FedEx). The database must be able to keep track of customers who ship items and customers who receive items; some customers may do both. Each package must be identifiable and trackable, so the database must be able to store the location of the package and its history of locations. Locations include trucks, planes, airports, and warehouses.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

6.24 Design a database for an airline. The database must keep track of customers and their reservations, flights and their status, seat assignments on individual flights, and the schedule and routing of future flights.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

Telegram: @uni_k

www.konkur.in

6.25 In Section 6.9.4, we represented a ternary relationship (repeated in Figure 6.29a) using binary relationships, as shown in Figure 6.29b. Consider the alternative shown in Figure 6.29c. Discuss the relative merits of these two alternative representations of a ternary relationship by binary relationships.

6.26 Design a generalization – specialization hierarchy for a motor vehicle sales company. The company sells motorcycles, passenger cars, vans, and buses. Justify your placement of attributes at each level of the hierarchy. Explain why they should not be placed at a higher or lower level.

6.27 Explain the distinction between disjoint and overlapping constraints.

6.28 Explain the distinction between total and partial constraints.

Tools

Many database systems provide tools for database design that support E-R diagrams. These tools help a designer create E-R diagrams, and they can automatically create corresponding tables in a database. See bibliographical notes of Chapter 1 for references to database-system vendors' web sites.

There are also several database-independent data modeling tools that support E-R diagrams and UML class diagrams.

Dia, which is a free diagram editor that runs on multiple platforms such as Linux and Windows, supports E-R diagrams and UML class diagrams. To represent entities with attributes, you can use either classes from the UML library or tables from the Database library provided by Dia, since the default E-R notation in Dia represents attributes as ovals. The free online diagram editor *LucidChart* allows you to create E-R diagrams with entities represented in the same ways as we do. To create relationships, we suggest you use diamonds from the Flowchart shape collection. *Draw.io* is another online diagram editor that supports E-R diagrams.

Commercial tools include IBM Rational Rose Modeler, Microsoft Visio, ERwin Data Modeler, Poseidon for UML, and SmartDraw.

Further Reading

The E-R data model was introduced by [Chen (1976)]. The Integration Definition for Information Modeling (IDEF1X) standard [NIST (1993)] released by the United States National Institute of Standards and Technology (NIST) defined standards for E-R diagrams. However, a variety of E-R notations are in use today.

[Thalheim (2000)] provides a detailed textbook coverage of research in E-R modeling.

As of 2018, the current UML version was 2.5, which was released in June 2015. See www.uml.org for more information on UML standards and tools.

Telegram: @uni_k

www.konkur.in

Bibliography

[Chen (1976)] P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, Volume 1, Number 1 (1976), pages 9–36.

[NIST (1993)] NIST, "Integration Definition for Information Modeling (IDEF1X)", Technical Report Federal Information Processing Standards Publication 184, National Institute of Standards and Technology (NIST) (1993).

[Thalheim (2000)] B. Thalheim, *Entity-Relationship Modeling: Foundations of Database Technology*, Springer Verlag (2000).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 7



Relational Database Design

In this chapter, we consider the problem of designing a schema for a relational database. Many of the issues in doing so are similar to design issues we considered in Chapter 6 using the E-R model.

In general, the goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. This is accomplished by designing schemas that are in an appropriate *normal form*. To determine whether a relation schema is in one of the desirable normal forms, we need information about the real-world enterprise that we are modeling with the database. Some of this information exists in a well-designed E-R diagram, but additional information about the enterprise may be needed as well.

In this chapter, we introduce a formal approach to relational database design based on the notion of functional dependencies. We then define normal forms in terms of functional dependencies and other types of data dependencies. First, however, we view the problem of relational design from the standpoint of the schemas derived from a given entity-relationship design.

7.1 Features of Good Relational Designs

Our study of entity-relationship design in Chapter 6 provides an excellent starting point for creating a relational database design. We saw in Section 6.7 that it is possible to generate a set of relation schemas directly from the E-R design. The goodness (or badness) of the resulting set of schemas depends on how good the E-R design was in the first place. Later in this chapter, we shall study precise ways of assessing the desirability of a collection of relation schemas. However, we can go a long way toward a good design using concepts we have already studied. For ease of reference, we repeat the schemas for the university database in Figure 7.1.

Suppose that we had started out when designing the university enterprise with the schema *in_dep*.

in_dep (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

Telegram: @uni_k

www.konkur.in

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

Figure 7.1 Database schema for the university example.

This represents the result of a natural join on the relations corresponding to *instructor* and *department*. This seems like a good idea because some queries can be expressed using fewer joins, until we think carefully about the facts about the university that led to our E-R design.

Let us consider the instance of the *in_dep* relation shown in Figure 7.2. Notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department. For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt.

It is important that all these tuples agree as to the budget amount since otherwise our database would be inconsistent. In our original design using *instructor* and *department*, we stored the amount of each budget exactly once. This suggests that using *in_dep* is a bad idea since it stores the budget amounts redundantly and runs the risk that some user might update the budget amount in one tuple but not all, and thus create inconsistency.

Even if we decided to live with the redundancy problem, there is still another problem with the *in_dep* schema. Suppose we are creating a new department in the university. In the alternative design above, we cannot represent directly the information concerning a department (*dept_name*, *building*, *budget*) unless that department has at least one instructor at the university. This is because tuples in the *in_dep* table require values for *ID*, *name*, and *salary*. This means that we cannot record information about the newly created department until the first instructor is hired for the new department. In the old design, the schema *department* can handle this, but under the revised design, we would have to create a tuple with a null value for *building* and *budget*. In some cases null values are troublesome, as we saw in our study of . However, if we decide that

Telegram: @uni_k

www.konkur.in

22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 7.2 The *in_dep* relation.

this is not a problem to us in this case, then we can proceed to use the revised design, though, as we noted, we would still have the redundancy problem.

7.1.1 Decomposition

The only way to avoid the repetition-of-information problem in the *in_dep* schema is to decompose it into two schemas (in this case, the *instructor* and *department* schemas). Later on in this chapter we shall present algorithms to decide which schemas are appropriate and which ones are not. In general, a schema that exhibits repetition of information may have to be decomposed into several smaller schemas.

Not all decompositions of schemas are helpful. Consider an extreme case in which all schemas consist of one attribute. No interesting relationships of any kind could be expressed. Now consider a less extreme case where we choose to decompose the *employee* schema (Section 6.8):

employee (*ID*, *name*, *street*, *city*, *salary*)

into the following two schemas:

employee1 (*ID*, *name*)
employee2 (*name*, *street*, *city*, *salary*)

The flaw in this decomposition arises from the possibility that the enterprise has two employees with the same name. This is not unlikely in practice, as many cultures have certain highly popular names. Each person would have a unique employee-id, which is why *ID* can serve as the primary key. As an example, let us assume two employees,

Telegram: @uni_k

www.konkur.in

both named Kim, work at the university and have the following tuples in the relation on schema *employee* in the original design:

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

Figure 7.3 shows these tuples, the resulting tuples using the schemas resulting from the decomposition, and the result if we attempted to regenerate the original tuples using a natural join. As we see in the figure, the two original tuples appear in the result along with two new tuples that incorrectly mix data values pertaining to the two employees named Kim. Although we have more tuples, we actually have less information in the following sense. We can indicate that a certain street, city, and salary pertain to someone named Kim, but we are unable to distinguish which of the Kims. Thus, our decomposition is unable to represent certain important facts about the university

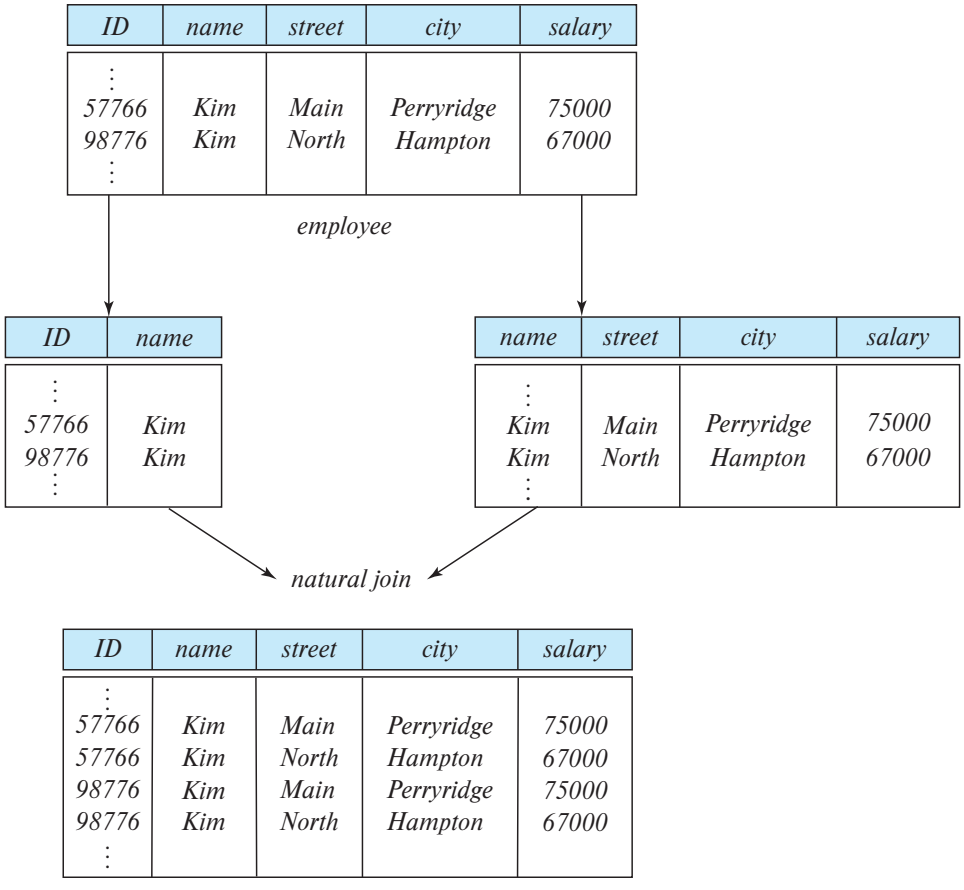


Figure 7.3 Loss of information via a bad decomposition.

Telegram: @uni_k

www.konkur.in

employees. We would like to avoid such decompositions. We shall refer to such decompositions as being **lossy decompositions**, and, conversely, to those that are not as **lossless decompositions**.

For the remainder of the text we shall insist that all decompositions should be lossless decompositions.

7.1.2 Lossless Decomposition

Let R be a relation schema and let R_1 and R_2 form a decomposition of R—that is, viewing R, R_1, and R_2 as sets of attributes, $R = R_1 \cup R_2$. We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with two relation schemas R_1 and R_2. Loss of information occurs if it is possible to have an instance of a relation $r(R)$ that includes information that cannot be represented if instead of the instance of $r(R)$ we must use instances of $r_1(R_1)$ and $r_2(R_2)$. More precisely, we say the decomposition is lossless if, for all legal (we shall formally define "legal" in Section 7.2.2.) database instances, relation r contains the same set of tuples as the result of the following SQL query:[1]

```
select *
from (select R1 from r)
     natural join
     (select R2 from r)
```

This is stated more succinctly in the relational algebra as:

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

In other words, if we project r onto R_1 and R_2, and compute the natural join of the projection results, we get back exactly r.

Conversely, a decomposition is lossy if when we compute the natural join of the projection results, we get a proper superset of the original relation. This is stated more succinctly in the relational algebra as:

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Let us return to our decomposition of the *employee* schema into *employee1* and *employee2* (Figure 7.3) and a case where two or more employees have the same name. The result of *employee1* natural join *employee2* is a superset of the original relation *employee*, but the decomposition is lossy since the join result has lost information about which employee identifiers correspond to which addresses and salaries.

[1]The definition of lossless is stated assuming that no attribute that appears on the left side of a functional dependency can have a null value. This is explored further in Exercise 7.10.

Telegram: @uni_k

www.konkur.in

It may seem counterintuitive that we have *more* tuples but *less* information, but that is indeed the case. The decomposed version is unable to represent the *absence* of a connection between a name and an address or salary, and absence of a connection is indeed information.

7.1.3 Normalization Theory

We are now in a position to define a general methodology for deriving a set of schemas each of which is in "good form"; that is, does not suffer from the repetition-of-information problem.

The method for designing a relational database is to use a process commonly known as **normalization**. The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is:

- Decide if a given relation schema is in "good form." There are a number of different forms (called *normal form*s), which we cover in Section 7.3.
- If a given relation schema is not in "good form," then we decompose it into a number of smaller relation schemas, each of which is in an appropriate normal form. The decomposition must be a lossless decomposition.

To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. The most common approach is to use **functional dependencies**, which we cover in Section 7.2.

7.2 Decomposition Using Functional Dependencies

A database models a set of entities and relationships in the real world. There are usually a variety of constraints (rules) on the data in the real world. For example, some of the constraints that are expected to hold in a university database are:

1. Students and instructors are uniquely identified by their ID.
2. Each student and instructor has only one name.
3. Each instructor and student is (primarily) associated with only one department.[2]
4. Each department has only one value for its budget, and only one associated building.

[2] An instructor, in most real universities, can be associated with more than one department, for example, via a joint appointment or in the case of adjunct faculty. Similarly, a student may have two (or more) majors or a minor. Our simplified university schema models only the primary department associated

Telegram: @uni_k

www.konkur.in

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a legal instance of a database is one where all the relation instances are legal instances.

7.2.1 Notational Conventions

In discussing algorithms for relational database design, we shall need to talk about arbitrary relations and their schema, rather than talking only about examples. Recalling our introduction to the relational model in Chapter 2, we summarize our notation here.

- In general, we use Greek letters for sets of attributes (e.g., α). We use an uppercase Roman letter to refer to a relation schema. We use the notation $r(R)$ to show that the schema R is for relation r.

  A relation schema is a set of attributes, but not all sets of attributes are schemas. When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema. A Roman letter is used when we wish to indicate that the set of attributes is definitely a schema.
- When a set of attributes is a superkey, we may denote it by K. A superkey pertains to a specific relation schema, so we use the terminology "K is a superkey for R."
- We use a lowercase name for relations. In our examples, these names are intended to be realistic (e.g., *instructor*), while in our definitions and algorithms, we use single letters, like r.
- The notation $r(R)$ thus refers to the relation r with schema R. When we write $r(R)$, we thus refer both to the relation and its schema.
- A relation, has a particular value at any given time; we refer to that as an instance and use the term "instance of r." When it is clear that we are talking about an instance, we may use simply the relation name (e.g., r).

For simplicity, we assume that attribute names have only one meaning within the database schema.

7.2.2 Keys and Functional Dependencies

Some of the most commonly used types of real-world constraints can be represented formally as keys (superkeys, candidate keys, and primary keys), or as functional dependencies, which we define below.

In Section 2.3, we defined the notion of a *superkey* as a set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation. We restate that definition here as follows: Given $r(R)$, a subset K of R is a **superkey** of $r(R)$ if, in any legal instance of $r(R)$, for all pairs t_1 and t_2 of tuples in the instance of r if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal instance of relation $r(R)$ may

Telegram: @uni_k

www.konkur.in

have the same value on attribute set K.[3] If no two tuples in r have the same value on K, then a K-value uniquely identifies a tuple in r.

Whereas a superkey is a set of attributes that uniquely identifies an entire tuple, a functional dependency allows us to express constraints that uniquely identify the values of certain attributes. Consider a relation schema $r(R)$, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of $r(R)$, we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t_1 and t_2 in the instance such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.
- We say that the functional dependency $\alpha \rightarrow \beta$ **holds** on schema $r(R)$ if, every legal instance of $r(R)$ satisfies the functional dependency.

Using the functional-dependency notation, we say that K *is a superkey for* $r(R)$ if the functional dependency $K \rightarrow R$ holds on $r(R)$. In other words, K is a superkey if, for every legal instance of $r(R)$, for every pair of tuples t_1 and t_2 from the instance, whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$ (i.e., $t_1 = t_2$).[4]

Functional dependencies allow us to express constraints that we cannot express with superkeys. In Section 7.1, we considered the schema:

$$in_dep\ (ID, name, salary, dept_name, building, budget)$$

in which the functional dependency $dept_name \rightarrow budget$ holds because for each department (identified by $dept_name$) there is a unique budget amount.

We denote the fact that the pair of attributes (ID, $dept_name$) forms a superkey for in_dep by writing:

$$ID, dept_name \rightarrow name, salary, building, budget$$

We shall use functional dependencies in two ways:

1. To test instances of relations to see whether they *satisfy* a given set F of functional dependencies.
2. To specify constraints on the set of legal relations. We shall thus concern ourselves with *only* those relation instances that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema $r(R)$ that satisfy a set F of functional dependencies, we say that F **holds** on $r(R)$.

[3] In our discussion of functional dependencies, we use equality (=) in the normal mathematical sense, not the three-valued-logic sense of SQL. Said differently, in discussing functional dependencies, we assume no null values.

[4] Note that we assume here that relations are sets. SQL deals with multisets, and a **primary key** declaration in SQL for a set of attributes K requires not only that $t_1 = t_2$ if $t_1[K] = t_2[K]$, but also that there be no duplicate tuples. SQL also requires that attributes in the set K cannot be assigned a *null* value.

Telegram: @uni_k

www.konkur.in

a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figure 7.4 Sample instance of relation r.

Let us consider the instance of relation r of Figure 7.4, to see which functional dependencies are satisfied. Observe that $A \rightarrow C$ is satisfied. There are two tuples that have an A value of a_1. These tuples have the same C value—namely, c_1. Similarly, the two tuples with an A value of a_2 have the same C value, c_2. There are no other pairs of distinct tuples that have the same A value. The functional dependency $C \rightarrow A$ is not satisfied, however. To see that it is not, consider the tuples $t_1 = (a_2, b_3, c_2, d_3)$ and $t_2 = (a_3, b_3, c_2, d_4)$. These two tuples have the same C values, c_2, but they have different A values, a_2 and a_3, respectively. Thus, we have found a pair of tuples t_1 and t_2 such that $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.

Some functional dependencies are said to be **trivial** because they are satisfied by all relations. For example, $A \rightarrow A$ is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally, we see that, for all tuples t_1 and t_2 such that $t_1[A] = t_2[A]$, it is the case that $t_1[A] = t_2[A]$. Similarly, $AB \rightarrow A$ is satisfied by all relations involving attribute A. In general, a functional dependency of the form $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

It is important to realize that an instance of a relation may satisfy some functional dependencies that are not required to hold on the relation's schema. In the instance of the *classroom* relation of Figure 7.5, we see that *room_number* $\rightarrow$ *capacity* is satisfied. However, we believe that, in the real world, two classrooms in different buildings can have the same room number but with different room capacity. Thus, it is possible, at some time, to have an instance of the *classroom* relation in which *room_number* $\rightarrow$ *capacity* is not satisfied. So, we would not include *room_number* $\rightarrow$ *capacity* in the set of

Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 7.5 An instance of the *classroom* relation.

Telegram: @uni_k

www.konkur.in

functional dependencies that hold on the schema for the *classroom* relation. However, we would expect the functional dependency *building*, *room_number* → *capacity* to hold on the *classroom* schema.

Because we assume that attribute names have only one meaning in the database schema, if we state that a functional dependency $\alpha \rightarrow \beta$ holds as a constraint on the database, then for any schema R such that $\alpha \subseteq R$ and $\beta \subseteq R$, $\alpha \rightarrow \beta$ must hold.

Given that a set of functional dependencies F holds on a relation $r(R)$, it may be possible to infer that certain other functional dependencies must also hold on the relation. For example, given a schema $r(A, B, C)$, if functional dependencies $A \rightarrow B$ and $B \rightarrow C$ hold on r, we can infer the functional dependency $A \rightarrow C$ must also hold on r. This is because, given any value of A, there can be only one corresponding value for B, and for that value of B, there can only be one corresponding value for C. We study in Section 7.4.1, how to make such inferences.

We shall use the notation F^+ to denote the **closure** of the set F, that is, the set of all functional dependencies that can be inferred given the set F. F^+ contains all of the functional dependencies in F.

7.2.3 Lossless Decomposition and Functional Dependencies

We can use functional dependencies to show when certain decompositions are lossless. Let R, R_1, R_2, and F be as above. R_1 and R_2 form a lossless decomposition of R if at least one of the following functional dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

In other words, if $R_1 \cap R_2$ forms a superkey for either R_1 or R_2, the decomposition of R is a lossless decomposition. We can use attribute closure to test efficiently for superkeys, as we have seen earlier.

To illustrate this, consider the schema

in_dep (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

that we decomposed in Section 7.1 into the *instructor* and *department* schemas:

instructor (*ID*, *name*, *dept_name*, *salary*)
department (*dept_name*, *building*, *budget*)

Consider the intersection of these two schemas, which is *dept_name*. We see that because *dept_name*→ *dept_name*, *building*, *budget*, the lossless-decomposition rule is satisfied.

Telegram: @uni_k

www.konkur.in

For the general case of decomposition of a schema into multiple schemas at once, the test for lossless decomposition is more complicated. See the Further Reading section at the end of this chapter for references on this topic.

While the test for binary decomposition is clearly a sufficient condition for lossless decomposition, it is a necessary condition only if all constraints are functional dependencies. We shall see other types of constraints later (in particular, a type of constraint called multivalued dependencies discussed in Section 7.6.1) that can ensure that a decomposition is lossless even if no functional dependencies are present.

Suppose we decompose a relation schema $r(R)$ into $r_1(R_1)$ and $r_2(R_2)$, where $R_1 \cap R_2 \rightarrow R_1$.[5] Then the following SQL constraints must be imposed on the decomposed schema to ensure their contents are consistent with the original schema.

- $R_1 \cap R_2$ is the primary key of r_1.
  This constraint enforces the functional dependency.
- $R_1 \cap R_2$ is a foreign key from r_2 referencing r_1.
  This constraint ensures that each tuple in r_2 has a matching tuple in r_1, without which it would not appear in the natural join of r_1 and r_2.

If r_1 or r_2 is decomposed further, as long as the decomposition ensures that all attributes in $R_1 \cap R_2$ are in one relation, the primary or foreign-key constraint on r_1 or r_2 would be inherited by that relation.

7.3 Normal Forms

As stated in Section 7.1.3, there are a number of different normal forms that are used in designing relational databases. In this section, we cover two of the most common ones.

7.3.1 Boyce–Codd Normal Form

One of the more desirable normal forms that we can obtain is **Boyce–Codd normal form** (**BCNF**). It eliminates all redundancy that can be discovered based on functional dependencies, though, as we shall see in Section 7.6, there may be other types of redundancy remaining.

7.3.1.1 Definition

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

[5] The case for $R_1 \cap R_2 \rightarrow R_2$ is symmetrical, and ignored.

Telegram: @uni_k

www.konkur.in

- $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e., $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

We have already seen in Section 7.1 an example of a relational schema that is not in BCNF:

$$in_dep\ (ID, name, salary, dept_name, building, budget)$$

The functional dependency $dept_name \rightarrow budget$ holds on *in_dep*, but *dept_name* is not a superkey (because a department may have a number of different instructors). In Section 7.1 we saw that the decomposition of *in_dep* into *instructor* and *department* is a better design. The *instructor* schema is in BCNF. All of the nontrivial functional dependencies that hold, such as:

$$ID \rightarrow name, dept_name, salary$$

include *ID* on the left side of the arrow, and *ID* is a superkey (actually, in this case, the primary key) for *instructor*. (In other words, there is no nontrivial functional dependency with any combination of *name*, *dept_name*, and *salary*, without *ID*, on the left side.) Thus, *instructor* is in BCNF.

Similarly, the *department* schema is in BCNF because all of the nontrivial functional dependencies that hold, such as:

$$dept_name \rightarrow building, budget$$

include *dept_name* on the left side of the arrow, and *dept_name* is a superkey (and the primary key) for *department*. Thus, *department* is in BCNF.

We now state a general rule for decomposing schemas that are not in BCNF. Let R be a schema that is not in BCNF. Then there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R. We replace R in our design with two schemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

In the case of *in_dep* above, $\alpha =$ *dept_name*, $\beta = \{$*building*, *budget*$\}$, and *in_dep* is replaced by

- $(\alpha \cup \beta)$ = (*dept_name*, *building*,*budget*)
- $(R - (\beta - \alpha))$ = (*ID*, *name*, *dept_name*, *salary*)

Telegram: @uni_k

www.konkur.in

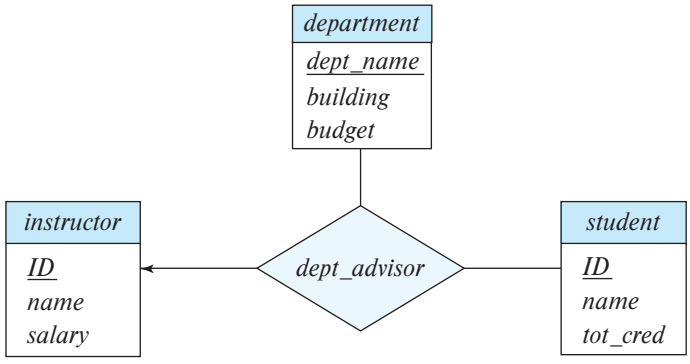


Figure 7.6 The *dept_advisor* relationship set.

In this example, it turns out that $\beta - \alpha = \beta$. We need to state the rule as we did so as to deal correctly with functional dependencies that have attributes that appear on both sides of the arrow. The technical reasons for this are covered later in Section 7.5.1.

When we decompose a schema that is not in BCNF, it may be that one or more of the resulting schemas are not in BCNF. In such cases, further decomposition is required, the eventual result of which is a set of BCNF schemas.

7.3.1.2 BCNF and Dependency Preservation

We have seen several ways in which to express database consistency constraints: primary-key constraints, functional dependencies, **check** constraints, assertions, and triggers. Testing these constraints each time the database is updated can be costly and, therefore, it is useful to design the database in a way that constraints can be tested efficiently. In particular, if testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low. We shall see that, in some cases, decomposition into BCNF can prevent efficient testing of certain functional dependencies.

To illustrate this, suppose that we make a small change to our university organization. In the design of Figure 6.15, a student may have only one advisor. This follows from the relationship set *advisor* being many-to-one from *student* to *advisor*. The "small" change we shall make is that an instructor can be associated with only a single department, and a student may have more than one advisor, but no more than one from a given department.[6]

One way to implement this change using the E-R design is by replacing the *advisor* relationship set with a ternary relationship set, *dept_advisor*, involving entity sets *instructor*, *student*, and *department* that is many-to-one from the pair {*student*, *instructor*} to *department* as shown in Figure 7.6. The E-R diagram specifies the constraint that

[6] Such an arrangement makes sense for students with a double

Telegram: @uni_k

www.konkur.in

"a student may have more than one advisor, but at most one corresponding to a given department."

With this new E-R diagram, the schemas for the *instructor*, *department*, and *student* relations are unchanged. However, the schema derived from the *dept_advisor* relationship set is now:

$$dept_advisor\ (s_ID,\ i_ID,\ dept_name)$$

Although not specified in the E-R diagram, suppose we have the additional constraint that "an instructor can act as advisor for only a single department."

Then, the following functional dependencies hold on *dept_advisor*:

$$i_ID \rightarrow dept_name$$
$$s_ID,\ dept_name \rightarrow i_ID$$

The first functional dependency follows from our requirement that "an instructor can act as an advisor for only one department." The second functional dependency follows from our requirement that "a student may have at most one advisor for a given department."

Notice that with this design, we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship. We see that *dept_advisor* is not in BCNF because *i_ID* is not a superkey. Following our rule for BCNF decomposition, we get:

$$(s_ID,\ i_ID)$$
$$(i_ID,\ dept_name)$$

Both the above schemas are BCNF. (In fact, you can verify that any schema with only two attributes is in BCNF by definition.)

Note, however, that in our BCNF design, there is no schema that includes all the attributes appearing in the functional dependency $s_ID,\ dept_name \rightarrow i_ID$. The only dependency that can be enforced on the individual decomposed relations is $ID \rightarrow dept_name$. The functional dependency $s_ID,\ dept_name \rightarrow i_ID$ can only be checked by computing the join of the decomposed relations.[7]

Because our design does not permit the enforcement of this functional dependency without a join, we say that our design is *not* **dependency preserving** (we provide a formal definition of dependency preservation in Section 7.4.4). Because dependency preservation is usually considered desirable, we consider another normal form, weaker than BCNF, that will allow us to preserve dependencies. That normal form is called the third normal form.[8]

[7] Technically, it is possible that a dependency whose attributes do not all appear in any one schema is still implicitly enforced, because of the presence of other dependencies that imply it logically. We address that case in Section 7.4.4.

[8] You may have noted that we skipped second normal form. It is of historical significance only and, in practice, one of third normal form or BCNF is always a better choice. We explore second normal form in Exercise 7.19. First normal form pertains to attribute domains, not decomposition. We discuss it in

Telegram: @uni_k

www.konkur.in

7.3.2 Third Normal Form

BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where α is a superkey. Third normal form (3NF) relaxes this constraint slightly by allowing certain nontrivial functional dependencies whose left side is not a superkey. Before we define 3NF, we recall that a candidate key is a minimal superkey—that is, a superkey no proper subset of which is also a superkey.

A relation schema R is in **third normal form** with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R.
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

Note that the third condition above does not say that a single candidate key must contain all the attributes in $\beta - \alpha$; each attribute A in $\beta - \alpha$ may be contained in a *different* candidate key.

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative in the 3NF definition seems rather unintuitive, and it is not obvious why it is useful. It represents, in some sense, a minimal relaxation of the BCNF conditions that helps ensure that every schema has a dependency-preserving decomposition into 3NF. Its purpose will become more clear later, when we study decomposition into 3NF.

Observe that any schema that satisfies BCNF also satisfies 3NF, since each of its functional dependencies would satisfy one of the first two alternatives. BCNF is therefore a more restrictive normal form than is 3NF.

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency $\alpha \rightarrow \beta$ that satisfies only the third alternative of the 3NF definition is not allowed in BCNF but is allowed in 3NF.[9]

Now, let us again consider the schema for the *dept_advisor* relation, which has the following functional dependencies:

$$i_ID \rightarrow dept_name$$
$$s_ID, dept_name \rightarrow i_ID$$

In Section 7.3.1.2, we argued that the functional dependency "$i_ID \rightarrow dept_name$" caused the *dept_advisor* schema not to be in BCNF. Note that here $\alpha = i_ID$, $\beta = dept_name$, and $\beta - \alpha = dept_name$. Since the functional dependency $s_ID, dept_name \rightarrow$

[9]These dependencies are examples of **transitive dependencies** (see Practice Exercise 7.18). The original definition of 3NF was in terms of transitive dependencies. The definition we use

Telegram: @uni_k

www.konkur.in

i_ID holds on *dept_advisor*, the attribute *dept_name* is contained in a candidate key and, therefore, *dept_advisor* is in 3NF.

We have seen the trade-off that must be made between BCNF and 3NF when there is no dependency-preserving BCNF design. These trade-offs are described in more detail in Section 7.3.3.

7.3.3 Comparison of BCNF and 3NF

Of the two normal forms for relational database schemas, 3NF and BCNF there are advantages to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation. Nevertheless, there are disadvantages to 3NF: We may have to use null values to represent some of the possible meaningful relationships among data items, and there is the problem of repetition of information.

Our goals of database design with functional dependencies are:

1. BCNF.
2. Losslessness.
3. Dependency preservation.

Since it is not always possible to satisfy all three, we may be forced to choose between BCNF and dependency preservation with 3NF.

It is worth noting that SQL does not provide a way of specifying functional dependencies, except for the special case of declaring superkeys by using the **primary key** or **unique** constraints. It is possible, although a little complicated, to write assertions that enforce a functional dependency (see Practice Exercise 7.9); unfortunately, currently no database system supports the complex assertions that are required to enforce arbitrary functional dependencies, and the assertions would be expensive to test. Thus even if we had a dependency-preserving decomposition, if we use standard SQL we can test efficiently only those functional dependencies whose left-hand side is a key.

Although testing functional dependencies may involve a join if the decomposition is not dependency preserving, if the database system supports materialized views, we could in principle reduce the cost by storing the join result as materialized view; however, this approach is feasible only if the database system supports **primary key** constraints or **unique** constraints on materialized views. On the negative side, there is a space and time overhead due to the materialized view, but on the positive side, the application programmer need not worry about writing code to keep redundant data consistent on updates; it is the job of the database system to maintain the materialized view, that is, keep it up to date when the database is updated. (In Section 16.5, we outline how a database system can perform materialized view maintenance efficiently.)

Unfortunately, most current database systems limit constraints on materialized views or do not support them at all. Even if such constraints are allowed, there is an additional requirement: the database must update the view and check the constraint

Telegram: @uni_k

www.konkur.in

immediately (as part of the same transaction) when an underlying relation is updated. Otherwise, a constraint violation may get detected well after the update has been performed and the transaction that caused the violation has committed.

In summary, even if we are not able to get a dependency-preserving BCNF decomposition, it is still preferable to opt for BCNF, since checking functional dependencies other than primary key constraints is difficult in SQL.

7.3.4 Higher Normal Forms

Using functional dependencies to decompose schemas may not be sufficient to avoid unnecessary repetition of information in certain cases. Consider a slight variation in the *instructor* entity-set definition in which we record with each instructor a set of children's names and a set of landline phone numbers that may be shared by multiple people. Thus, *phone_number* and *child_name* would be multivalued attributes and, following our rules for generating schemas from an E-R design, we would have two schemas, one for each of the multivalued attributes, *phone_number* and *child_name*:

(*ID*, *child_name*)
(*ID*, *phone_number*)

If we were to combine these schemas to get

(*ID*, *child_name*, *phone_number*)

we would find the result to be in BCNF because no nontrivial functional dependencies hold. As a result we might think that such a combination is a good idea. However, such a combination is a bad idea, as we can see by considering the example of an instructor with two children and two phone numbers. For example, let the instructor with *ID* 99999 have two children named "David" and "William" and two phone numbers, 512-555-1234 and 512-555-4321. In the combined schema, we must repeat the phone numbers once for each dependent:

(99999, David, 512-555-1234)
(99999, David, 512-555-4321)
(99999, William, 512-555-1234)
(99999, William, 512-555-4321)

If we did not repeat the phone numbers, and we stored only the first and last tuples, we would have recorded the dependent names and the phone numbers, but the resultant tuples would imply that David corresponded to 512-555-1234, while William corresponded to 512-555-4321. This would be incorrect.

Because normal forms based on functional dependencies are not sufficient to deal with situations like this, other dependencies and normal forms have been defined. We cover these in Section 7.6 and Section 7.7.

Telegram: @uni_k

www.konkur.in

7.4 Functional-Dependency Theory

We have seen in our examples that it is useful to be able to reason systematically about functional dependencies as part of a process of testing schemas for BCNF or 3NF.

7.4.1 Closure of a Set of Functional Dependencies

We shall see that, given a set F of functional dependencies on a schema, we can prove that certain other functional dependencies also hold on the schema. We say that such functional dependencies are "logically implied" by F. When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we need to consider *all* functional dependencies that hold on the schema.

More formally, given a relation schema $r(R)$, a functional dependency f on R is **logically implied** by a set of functional dependencies F on R if every instance of a relation $r(R)$ that satisfies F also satisfies f.

Suppose we are given a relation schema $r(A, B, C, G, H, I)$ and the set of functional dependencies:

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

The functional dependency:

$$A \rightarrow H$$

is logically implied. That is, we can show that, whenever a relation instance satisfies our given set of functional dependencies, $A \rightarrow H$ must also be satisfied by that relation instance. Suppose that t_1 and t_2 are tuples such that:

$$t_1[A] = t_2[A]$$

Since we are given that $A \rightarrow B$, it follows from the definition of functional dependency that:

$$t_1[B] = t_2[B]$$

Then, since we are given that $B \rightarrow H$, it follows from the definition of functional dependency that:

$$t_1[H] = t_2[H]$$

Telegram: @uni_k

www.konkur.in

Therefore, we have shown that, whenever t_1 and t_2 are tuples such that $t_1[A] = t_2[A]$, it must be that $t_1[H] = t_2[H]$. But that is exactly the definition of $A \rightarrow H$.

Let F be a set of functional dependencies. The **closure** of F, denoted by F^+, is the set of all functional dependencies logically implied by F. Given F, we can compute F^+ directly from the formal definition of functional dependency. If F were large, this process would be lengthy and difficult. Such a computation of F^+ requires arguments of the type just used to show that $A \rightarrow H$ is in the closure of our example set of dependencies.

Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies. In the rules that follow, we use Greek letters (α, β, γ, ...) for sets of attributes and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use $\alpha\beta$ to denote $\alpha \cup \beta$.

We can use the following three rules to find logically implied functional dependencies. By applying these rules *repeatedly*, we can find all of F^+, given F. This collection of rules is called **Armstrong's axioms** in honor of the person who first proposed it.

- **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Armstrong's axioms are **sound**, because they do not generate any incorrect functional dependencies. They are **complete**, because, for a given set F of functional dependencies, they allow us to generate all F^+. The Further Reading section provides references for proofs of soundness and completeness.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of F^+. To simplify matters further, we list additional rules. It is possible to use Armstrong's axioms to prove that these rules are sound (see Practice Exercise 7.4, Practice Exercise 7.5, and Exercise 7.27).

- **Union rule**. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule**. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Let us apply our rules to the example of schema $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. We list several members of F^+ here:

- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds than it was to argue directly from the definitions, as we did earlier in this section.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.

Telegram: @uni_k

www.konkur.in


```
F+ = F
apply the reflexivity rule /* Generates all trivial dependencies */
repeat
    for each functional dependency f in F+
        apply the augmentation rule on f
        add the resulting functional dependencies to F+
    for each pair of functional dependencies f1 and f2 in F+
        if f1 and f2 can be combined using transitivity
            add the resulting functional dependency to F+
until F+ does not change any further
```

Figure 7.7 A procedure to compute F^+.

- $AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudotransitivity rule implies that $AG \rightarrow I$ holds.

  Another way of finding that $AG \rightarrow I$ holds is as follows: We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow CG$. Applying the transitivity rule to this dependency and $CG \rightarrow I$, we infer $AG \rightarrow I$.

Figure 7.7 shows a procedure that demonstrates formally how to use Armstrong's axioms to compute F^+. In this procedure, when a functional dependency is added to F^+, it may be already present, and in that case there is no change to F^+. We shall see an alternative way of computing F^+ in Section 7.4.2.

The left-hand and right-hand sides of a functional dependency are both subsets of R. Since a set of size n has 2^n subsets, there are a total of $2^n \times 2^n = 2^{2n}$ possible functional dependencies, where n is the number of attributes in R. Each iteration of the repeat loop of the procedure, except the last iteration, adds at least one functional dependency to F^+. Thus, the procedure is guaranteed to terminate, though it may be very lengthy.

7.4.2 Closure of Attribute Sets

We say that an attribute B is **functionally determined** by α if $\alpha \rightarrow B$. To test whether a set α is a superkey, we must devise an algorithm for computing the set of attributes functionally determined by α. One way of doing this is to compute F^+, take all functional dependencies with α as the left-hand side, and take the union of the right-hand sides of all such dependencies. However, doing so can be expensive, since F^+ can be large.

An efficient algorithm for computing the set of attributes functionally determined by α is useful not only for testing whether α is a superkey, but also for several other tasks, as we shall see later in this section.

Telegram: @uni_k

www.konkur.in

Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set F of functional dependencies the closure of α under F; we denote it by α^+. Figure 7.8 shows an algorithm, written in pseudocode, to compute α^+. The input is a set F of functional dependencies and the set α of attributes. The output is stored in the variable *result*.

To illustrate how the algorithm works, we shall use it to compute $(AG)^+$ with the functional dependencies defined in Section 7.4.1. We start with $result = AG$. The first time that we execute the **repeat** loop to test each functional dependency, we find that:

- $A \rightarrow B$ causes us to include B in *result*. To see this fact, we observe that $A \rightarrow B$ is in F, $A \subseteq result$ (which is AG), so $result := result \cup B$.
- $A \rightarrow C$ causes *result* to become $ABCG$.
- $CG \rightarrow H$ causes *result* to become $ABCGH$.
- $CG \rightarrow I$ causes *result* to become $ABCGHI$.

The second time that we execute the **repeat** loop, no new attributes are added to *result*, and the algorithm terminates.

Let us see why the algorithm of Figure 7.8 is correct. The first step is correct because $\alpha \rightarrow \alpha$ always holds (by the reflexivity rule). We claim that, for any subset β of *result*, $\alpha \rightarrow \beta$. Since we start the **repeat** loop with $\alpha \rightarrow result$ being true, we can add γ to *result* only if $\beta \subseteq result$ and $\beta \rightarrow \gamma$. But then $result \rightarrow \beta$ by the reflexivity rule, so $\alpha \rightarrow \beta$ by transitivity. Another application of transitivity shows that $\alpha \rightarrow \gamma$ (using $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$). The union rule implies that $\alpha \rightarrow result \cup \gamma$, so α functionally determines any new result generated in the **repeat** loop. Thus, any attribute returned by the algorithm is in α^+.

It is easy to see that the algorithm finds all of α^+. Consider an attribute A in α^+ that is not yet in *result* at any point during the execution. There must be a way to prove that $result \rightarrow A$ using the axioms. Either $result \rightarrow A$ is in F itself (making the proof trivial and ensuring A is added to *result*) or there must a proof step using transitivity to show

```
result := α;
repeat
    for each functional dependency β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
until (result does not change)
```

Figure 7.8 An algorithm to compute α^+, the closure of α under F.

Telegram: @uni_k

www.konkur.in

for some attribute B that $result \rightarrow B$. If it happens that $A = B$, then we have shown that A is added to *result*. If not, $B \neq A$ is added. Then repeating this argument, we see that A must eventually be added to *result*.

It turns out that, in the worst case, this algorithm may take an amount of time quadratic in the size of F. There is a faster (although slightly more complex) algorithm that runs in time linear in the size of F; that algorithm is presented as part of Practice Exercise 7.8.

There are several uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α^+ and check if α^+ contains all attributes in R.
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), by checking if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if it contains β. This test is particularly useful, as we shall see later in this chapter.
- It gives us an alternative way to compute F^+: For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

7.4.3 Canonical Cover

Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies, that is, all the functional dependencies in F are satisfied in the new database state.

The system must roll back the update if it violates any functional dependencies in the set F.

We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set. Any database that satisfies the simplified set of functional dependencies also satisfies the original set, and vice versa, since the two sets have the same closure. However, the simplified set is easier to test. We shall see how the simplified set can be constructed in a moment. First, we need some definitions.

An attribute of a functional dependency is said to be **extraneous** if we can remove it without changing the closure of the set of functional dependencies.

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint. For example, if we have $AB \rightarrow C$ and remove B, we get the possibly stronger result $A \rightarrow C$. It may be stronger because $A \rightarrow C$ logically implies $AB \rightarrow C$, but $AB \rightarrow C$ does not, on its own, logically imply $A \rightarrow C$. But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from $AB \rightarrow C$ safely. For example, suppose that the set

Telegram: @uni_k

www.konkur.in

$F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$. Then we can show that F logically implies $A \rightarrow C$, making B extraneous in $AB \rightarrow C$.

- Removing an attribute from the right side of a functional dependency could make it a weaker constraint. For example, if we have $AB \rightarrow CD$ and remove C, we get the possibly weaker result $AB \rightarrow D$. It may be weaker because using just $AB \rightarrow D$, we can no longer infer $AB \rightarrow C$. But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely. For example, suppose that $F = \{AB \rightarrow CD, A \rightarrow C\}$. Then we can show that even after replacing $AB \rightarrow CD$ by $AB \rightarrow D$, we can still infer $AB \rightarrow C$ and thus $AB \rightarrow CD$.

The formal definition of **extraneous attributes** is as follows: Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.

- **Removal from the left side:** Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- **Removal from the right side:** Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

Beware of the direction of the implications when using the definition of extraneous attributes: If you reverse the statement, the implication will *always* hold. That is, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ always logically implies F, and also F always logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$.

Here is how we can test efficiently if an attribute is extraneous. Let R be the relation schema, and let F be the given set of functional dependencies that hold on R. Consider an attribute A in a dependency $\alpha \rightarrow \beta$.

- If $A \in \beta$, to check if A is extraneous, consider the set

  $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$

  and check if $\alpha \rightarrow A$ can be inferred from F'. To do so, compute α^+ (the closure of α) under F'; if α^+ includes A, then A is extraneous in β.
- If $A \in \alpha$, to check if A is extraneous, let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$ can be inferred from F. To do so, compute γ^+ (the closure of γ) under F; if γ^+ includes all attributes in β, then A is extraneous in α.

For example, suppose F contains $AB \rightarrow CD$, $A \rightarrow E$, and $E \rightarrow C$. To check if C is extraneous in $AB \rightarrow CD$, we compute the attribute closure of AB under $F' = \{AB \rightarrow D$, $A \rightarrow E$, $E \rightarrow C\}$. The closure is $ABCDE$, which includes CD, so we infer that C is extraneous.

Telegram: @uni_k

www.konkur.in

$F_c = F$
repeat
  Use the union rule to replace any dependencies in F_c of the form
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous
    attribute either in α or in β.
    /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.
until (F_c does not change)

Figure 7.9 Computing canonical cover.

Having defined the concept of extraneous attributes, we can explain how we can construct a simplified set of functional dependencies equivalent to a given set of functional dependencies.

A **canonical cover** F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F. Furthermore, F_c must have the following properties:

- No functional dependency in F_c contains an extraneous attribute.
- Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

A canonical cover for a set of functional dependencies F can be computed as described in Figure 7.9. It is important to note that when checking if an attribute is extraneous, the check uses the dependencies in the current value of F_c, and **not** the dependencies in F. If a functional dependency contains only one attribute in its right-hand side, for example $A \rightarrow C$, and that attribute is found to be extraneous, we would get a functional dependency with an empty right-hand side. Such functional dependencies should be deleted.

Since the algorithm permits a choice of any extraneous attribute, it is possible that there may be several possible canonical covers for a given F. Any such F_c is equally acceptable. Any canonical cover of F, F_c, can be shown to have the same closure as F; hence, testing whether F_c is satisfied is equivalent to testing whether F is satisfied. However, F_c is minimal in a certain sense—it does not contain extraneous attributes, and it combines functional dependencies with the same left side. It is cheaper to test F_c than it is to test F itself.

We now consider an example. Assume we are given the following set F of functional dependencies on schema (A, B, C):

Telegram: @uni_k

www.konkur.in

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

Let us compute a canonical cover for F.

- There are two functional dependencies with the same set of attributes on the left side of the arrow:

  $$A \rightarrow BC$$
  $$A \rightarrow B$$

  We combine these functional dependencies into $A \rightarrow BC$.
- A is extraneous in $AB \rightarrow C$ because F logically implies $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. This assertion is true because $B \rightarrow C$ is already in our set of functional dependencies.
- C is extraneous in $A \rightarrow BC$, since $A \rightarrow BC$ is logically implied by $A \rightarrow B$ and $B \rightarrow C$.

Thus, our canonical cover is:

$$A \rightarrow B$$
$$B \rightarrow C$$

Given a set F of functional dependencies, it may be that an entire functional dependency in the set is extraneous, in the sense that dropping it does not change the closure of F. We can show that a canonical cover F_c of F contains no such extraneous functional dependency. Suppose that, to the contrary, there were such an extraneous functional dependency in F_c. The right-side attributes of the dependency would then be extraneous, which is not possible by the definition of canonical covers.

As we noted earlier, a canonical cover might not be unique. For instance, consider the set of functional dependencies $F = \{A \rightarrow BC, B \rightarrow AC$, and $C \rightarrow AB\}$. If we apply the test for extraneous attributes to $A \rightarrow BC$, we find that both B and C are extraneous under F. However, it is incorrect to delete both! The algorithm for finding the canonical cover picks one of the two and deletes it. Then,

1. If C is deleted, we get the set $F' = \{A \rightarrow B, B \rightarrow AC$, and $C \rightarrow AB\}$. Now, B is not extraneous on the right side of $A \rightarrow B$ under F'. Continuing the algorithm, we find A and B are extraneous in the right side of $C \rightarrow AB$, leading to two choices of canonical cover:

Telegram: @uni_k

www.konkur.in


```
compute F+;
for each schema Ri in D do
    begin
        Fi : = the restriction of F+ to Ri;
    end
F' := ∅
for each restriction Fi do
    begin
        F' = F' ∪ Fi
    end
compute F'+;
if (F'+ = F+) then return (true)
                else return (false);
```

Figure 7.10 Testing for dependency preservation.

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}.$$

2. If B is deleted, we get the set $\{A \rightarrow C, B \rightarrow AC$, and $C \rightarrow AB\}$. This case is symmetrical to the previous case, leading to two more choices of canonical cover:

$$F_c = \{A \rightarrow C, C \rightarrow B, \text{ and } B \rightarrow A\}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, \text{ and } C \rightarrow AB\}.$$

As an exercise, can you find one more canonical cover for F?

7.4.4 Dependency Preservation

Using the theory of functional dependencies, there is a way to describe dependency preservation that is simpler than the ad hoc approach we used in Section 7.3.1.2.

Let F be a set of functional dependencies on a schema R, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. The **restriction of F to R_i** is the set F_i of all functional dependencies in F^+ that include *only* attributes of R_i. Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

Note that the definition of restriction uses all dependencies in F^+, not just those in F. For instance, suppose $F = \{A \rightarrow B, B \rightarrow C\}$, and we have a decomposition into AC and AB. The restriction of F to AC includes $A \rightarrow C$, since $A \rightarrow C$ is in F^+, even though it is not in F.

Telegram: @uni_k

www.konkur.in

The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let $F' = F_1 \cup F_2 \cup \cdots \cup F_n$. F' is a set of functional dependencies on schema R, but, in general, $F' \neq F$. However, even if $F' \neq F$, it may be that $F'^+ = F^+$. If the latter is true, then every dependency in F is logically implied by F', and, if we verify that F' is satisfied, we have verified that F is satisfied. We say that a decomposition having the property $F'^+ = F^+$ is a **dependency-preserving decomposition**.

Figure 7.10 shows an algorithm for testing dependency preservation. The input is a set $D = \{R_1, R_2, \ldots, R_n\}$ of decomposed relation schemas, and a set F of functional dependencies. This algorithm is expensive since it requires computation of F^+. Instead of applying the algorithm of Figure 7.10, we consider two alternatives.

First, note that if each member of F can be tested on one of the relations of the decomposition, then the decomposition is dependency preserving. This is an easy way to show dependency preservation; however, it does not always work. There are cases where, even though the decomposition is dependency preserving, there is a dependency in F that cannot be tested in any one relation in the decomposition. Thus, this alternative test can be used only as a sufficient condition that is easy to check; if it fails we cannot conclude that the decomposition is not dependency preserving; instead we will have to apply the general test.

We now give a second alternative test for dependency preservation that avoids computing F^+. We explain the intuition behind the test after presenting the test. The test applies the following procedure to each $\alpha \rightarrow \beta$ in F.

```
result = α
repeat
    for each R_i in the decomposition
        t = (result ∩ R_i)^+ ∩ R_i
        result = result ∪ t
until (result does not change)
```

The attribute closure here is under the set of functional dependencies F. If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved. The decomposition is dependency preserving if and only if the procedure shows that all the dependencies in F are preserved.

The two key ideas behind the preceding test are as follows:

- The first idea is to test each functional dependency $\alpha \rightarrow \beta$ in F to see if it is preserved in F' (where F' is as defined in Figure 7.10). To do so, we compute the closure of α under F'; the dependency is preserved exactly when the closure includes β. The decomposition is dependency preserving if (and only if) all the dependencies in F are found to be preserved.

Telegram: @uni_k

www.konkur.in

- The second idea is to use a modified form of the attribute-closure algorithm to compute closure under F', without actually first computing F'. We wish to avoid computing F' since computing it is quite expensive. Note that F' is the union of all F_i, where F_i is the restriction of F on R_i. The algorithm computes the attribute closure of $(result \cap R_i)$ with respect to F, intersects the closure with R_i, and adds the resultant set of attributes to *result*; this sequence of steps is equivalent to computing the closure of *result* under F_i. Repeating this step for each i inside the while loop gives the closure of *result* under F'.

  To understand why this modified attribute-closure approach works correctly, we note that for any $\gamma \subseteq R_i$, $\gamma \rightarrow \gamma^+$ is a functional dependency in F^+, and $\gamma \rightarrow \gamma^+ \cap R_i$ is a functional dependency that is in F_i, the restriction of F^+ to R_i. Conversely, if $\gamma \rightarrow \delta$ were in F_i, then δ would be a subset of $\gamma^+ \cap R_i$.

This test takes polynomial time, instead of the exponential time required to compute F^+.

7.5 Algorithms for Decomposition Using Functional Dependencies

Real-world database schemas are much larger than the examples that fit in the pages of a book. For this reason, we need algorithms for the generation of designs that are in appropriate normal form. In this section, we present algorithms for BCNF and 3NF.

7.5.1 BCNF Decomposition

The definition of BCNF can be used directly to test if a relation is in BCNF. However, computation of F^+ can be a tedious task. We first describe simplified tests for verifying if a relation is in BCNF. If a relation is not in BCNF, it can be decomposed to create relations that are in BCNF. Later in this section, we describe an algorithm to create a lossless decomposition of a relation, such that the decomposition is in BCNF.

7.5.1.1 Testing for BCNF

Testing of a relation schema R to see if it satisfies BCNF can be simplified in some cases:

- To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute α^+ (the attribute closure of α), and verify that it includes all attributes of R; that is, it is a superkey for R.
- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than check all dependencies in F^+.

  We can show that if none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF, either.

Telegram: @uni_k

www.konkur.in


```
result := {R};
done := false;
while (not done) do
   if (there is a schema R_i in result that is not in BCNF)
     then begin
             let α → β be a nontrivial functional dependency that holds
             on R_i such that α+ does not contain R_i and α ∩ β = ∅;
             result := (result − R_i) ∪ (R_i − β) ∪ ( α, β);
          end
     else done := true;
```

Figure 7.11 BCNF decomposition algorithm.

Unfortunately, the latter procedure does not work when a relation schema is decomposed. That is, it *does not* suffice to use F when we test a relation schema R_i, in a decomposition of R, for violation of BCNF. For example, consider relation schema (A, B, C, D, E), with functional dependencies F containing $A \rightarrow B$ and $BC \rightarrow D$. Suppose this were decomposed into (A, B) and (A, C, D, E). Now, neither of the dependencies in F contains only attributes from (A, C, D, E), so we might be misled into thinking that it is in BCNF. In fact, there is a dependency $AC \rightarrow D$ in F^+ (which can be inferred using the pseudotransitivity rule from the two dependencies in F) that shows that (A, C, D, E) is not in BCNF. Thus, we may need a dependency that is in F^+, but is not in F, to show that a decomposed relation is not in BCNF.

An alternative BCNF test is sometimes easier than computing every dependency in F^+. To check if a relation schema R_i in a decomposition of R is in BCNF, we apply this test:

- For every subset α of attributes in R_i, check that α^+ (the attribute closure of α under F) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.

If the condition is violated by some set of attributes α in R_i, consider the following functional dependency, which can be shown to be present in F^+:

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i.$$

This dependency shows that R_i violates BCNF.

7.5.1.2 BCNF Decomposition Algorithm

We are now able to state a general method to decompose a relation schema so as to satisfy BCNF. Figure 7.11 shows an algorithm for this task. If R is not in BCNF, we can decompose R into a collection of BCNF schemas $R_1, R_2, \ldots, R_n$ by the algorithm.

Telegram: @uni_k

www.konkur.in

The algorithm uses dependencies that demonstrate violation of BCNF to perform the decomposition.

The decomposition that the algorithm generates is not only in BCNF, but is also a lossless decomposition. To see why our algorithm generates only lossless decompositions, we note that, when we replace a schema R_i with $(R_i - \beta)$ and (α, β), the dependency $\alpha \rightarrow \beta$ holds, and $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

If we did not require $\alpha \cap \beta = \emptyset$, then those attributes in $\alpha \cap \beta$ would not appear in the schema $(R_i - \beta)$, and the dependency $\alpha \rightarrow \beta$ would no longer hold.

It is easy to see that our decomposition of *in_dep* in Section 7.3.1 would result from applying the algorithm. The functional dependency *dept_name* $\rightarrow$ *building*, *budget* satisfies the $\alpha \cap \beta = \emptyset$ condition and would therefore be chosen to decompose the schema.

The **BCNF decomposition algorithm** takes time exponential to the size of the initial schema, since the algorithm for checking whether a relation in the decomposition satisfies BCNF can take exponential time. There is an algorithm that can compute a BCNF decomposition in polynomial time; however, the algorithm may "overnormalize," that is, decompose a relation unnecessarily.

As a longer example of the use of the BCNF decomposition algorithm, suppose we have a database design using the *class* relation, whose schema is as shown below:

class (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

The set of functional dependencies that we need to hold on this schema are:

course_id $\rightarrow$ *title*, *dept_name*, *credits*
building, *room_number* $\rightarrow$ *capacity*
course_id, *sec_id*, *semester*, *year* $\rightarrow$ *building*, *room_number*, *time_slot_id*

A candidate key for this schema is {*course_id*, *sec_id*, *semester*, *year*}.

We can apply the algorithm of Figure 7.11 to the *class* example as follows:

- The functional dependency:

  course_id $\rightarrow$ *title*, *dept_name*, *credits*

  holds, but *course_id* is not a superkey. Thus, *class* is not in BCNF. We replace *class* with two relations with the following schemas:

  course (*course_id*, *title*, *dept_name*, *credits*)
  class-1 (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number* *capacity*, *time_slot_id*)

Telegram: @uni_k

www.konkur.in

The only nontrivial functional dependencies that hold on *course* include *course_id* on the left side of the arrow. Since *course_id* is a superkey for *course*, *course* is in BCNF.

- A candidate key for *class-1* is {*course_id*, *sec_id*, *semester*, *year*}. The functional dependency:

$$building, room_number \rightarrow capacity$$

holds on *class-1*, but {*building*, *room_number*} is not a superkey for *class-1*. We replace *class-1* two relations with the following schemas:

$$classroom\ (building, room_number, capacity)$$
$$section\ (course_id, sec_id, semester, year, building, room_number, time_slot_id)$$

These two schemas are in BCNF.

Thus, the decomposition of *class* results in the three relation schemas *course*, *classroom*, and *section*, each of which is in BCNF. These correspond to the schemas that we have used in this and previous chapters. You can verify that the decomposition is lossless and dependency preserving.

7.5.2 3NF Decomposition

Figure 7.12 shows an algorithm for finding a dependency-preserving, lossless decomposition into 3NF. The set of dependencies F_c used in the algorithm is a canonical cover for F. Note that the algorithm considers the set of schemas R_j, $j = 1, 2, \ldots, i$; initially $i = 0$, and in this case the set is empty.

Let us apply this algorithm to our example of *dept_advisor* from Section 7.3.2, where we showed that:

$$dept_advisor\ (s_ID, i_ID, dept_name)$$

is in 3NF even though it is not in BCNF. The algorithm uses the following functional dependencies in F:

$$f_1: i_ID \rightarrow dept_name$$
$$f_2: s_ID, dept_name \rightarrow i_ID$$

There are no extraneous attributes in any of the functional dependencies in F, so F_c contains f_1 and f_2. The algorithm then generates as R_1 the schema, (*i_ID dept_name*), and as R_2 the schema (*s_ID*, *dept_name*, *i_ID*). The algorithm then finds that R_2 contains a candidate key, so no further relation schema is created.

Telegram: @uni_k

www.konkur.in


```
let F_c be a canonical cover for F;
i := 0;
for each functional dependency α → β in F_c
    i := i + 1;
    R_i := α β;
if none of the schemas R_j, j = 1, 2, ..., i contains a candidate key for R
  then
    i := i + 1;
    R_i := any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        R_j := R_i;
        i := i - 1;
until no more R_j s can be deleted
return (R_1, R_2, ..., R_i)
```

Figure 7.12 Dependency-preserving, lossless decomposition into 3NF.

The resultant set of schemas can contain redundant schemas, with one schema R_k containing all the attributes of another schema R_j. For example, R_2 above contains all the attributes from R_1. The algorithm deletes all such schemas that are contained in another schema. Any dependencies that could be tested on an R_j that is deleted can also be tested on the corresponding relation R_k, and the decomposition is lossless even if R_j is deleted.

Now let us consider again the schema of the *class* relation of Section 7.5.1.2 and apply the **3NF decomposition algorithm**. The set of functional dependencies we listed there happen to be a canonical cover. As a result, the algorithm gives us the same three schemas *course, classroom*, and *section*.

The preceding example illustrates an interesting property of the 3NF algorithm. Sometimes, the result is not only in 3NF, but also in BCNF. This suggests an alternative method of generating a BCNF design. First use the 3NF algorithm. Then, for any schema in the 3NF design that is not in BCNF, decompose using the BCNF algorithm. If the result is not dependency-preserving, revert to the 3NF design.

7.5.3 Correctness of the 3NF Algorithm

The 3NF algorithm ensures the preservation of dependencies by explicitly building a schema for each dependency in a canonical cover. It ensures that the decomposition is a

Telegram: @uni_k

www.konkur.in

lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. Practice Exercise 7.16 provides some insight into the proof that this suffices to guarantee a lossless decomposition.

This algorithm is also called the **3NF synthesis algorithm**, since it takes a set of dependencies and adds one schema at a time, instead of decomposing the initial schema repeatedly. The result is not uniquely defined, since a set of functional dependencies can have more than one canonical cover. The algorithm may decompose a relation even if it is already in 3NF; however, the decomposition is still guaranteed to be in 3NF.

To see that the algorithm produces a 3NF design, consider a schema R_i in the decomposition. Recall that when we test for 3NF it suffices to consider functional dependencies whose right-hand side consists of a single attribute. Therefore, to see that R_i is in 3NF you must convince yourself that any functional dependency $\gamma \rightarrow B$ that holds on R_i satisfies the definition of 3NF. Assume that the dependency that generated R_i in the synthesis algorithm is $\alpha \rightarrow \beta$. B must be in α or β, since B is in R_i and $\alpha \rightarrow \beta$ generated R_i. Let us consider the three possible cases:

- B is in both α and β. In this case, the dependency $\alpha \rightarrow \beta$ would not have been in F_c since B would be extraneous in β. Thus, this case cannot hold.
- B is in β but not α. Consider two cases:
  - γ is a superkey. The second condition of 3NF is satisfied.
  - γ is not a superkey. Then α must contain some attribute not in γ. Now, since $\gamma \rightarrow B$ is in F^+, it must be derivable from F_c by using the attribute closure algorithm on γ. The derivation could not have used $\alpha \rightarrow \beta$, because if it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey. Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha\beta$, and γ cannot contain B because $\gamma \rightarrow B$ is nontrivial). This would imply that B is extraneous in the right-hand side of $\alpha \rightarrow \beta$, which is not possible since $\alpha \rightarrow \beta$ is in the canonical cover F_c. Thus, if B is in β, then γ must be a superkey, and the second condition of 3NF must be satisfied.
- B is in α but not β.
  Since α is a candidate key, the third alternative in the definition of 3NF is satisfied.

Interestingly, the algorithm we described for decomposition into 3NF can be implemented in polynomial time, even though testing a given schema to see if it satisfies 3NF is NP-hard (which means that it is very unlikely that a polynomial-time algorithm will ever be invented for this task).

Telegram: @uni_k

www.konkur.in

7.6 Decomposition Using Multivalued Dependencies

Some relation schemas, even though they are in BCNF, do not seem to be sufficiently normalized, in the sense that they still suffer from the problem of repetition of information. Consider a variation of the university organization where an instructor may be associated with multiple departments, and we have a relation:

$$inst\ (ID, dept_name, name, street, city)$$

The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency

$$ID \rightarrow name, street, city$$

and because *ID* is not a key for *inst*.

Further assume that an instructor may have several addresses (say, a winter home and a summer home). Then, we no longer wish to enforce the functional dependency "$ID \rightarrow street, city$", though, we still want to enforce "$ID \rightarrow name$" (i.e., the university is not dealing with instructors who operate under multiple aliases!). Following the BCNF decomposition algorithm, we obtain two schemas:

$$r_1\ (ID, name)$$
$$r_2\ (ID, dept_name, street, city)$$

Both of these are in BCNF (recall that an instructor can be associated with multiple departments and a department may have several instructors, and therefore, neither "$ID \rightarrow dept_name$" nor "$dept_name \rightarrow ID$" hold).

Despite r_2 being in BCNF, there is redundancy. We repeat the address information of each residence of an instructor once for each department with which the instructor is associated. We could solve this problem by decomposing r_2 further into:

$$r_{21}\ (dept_name, ID)$$
$$r_{22}\ (ID, street, city)$$

but there is no constraint that leads us to do this.

To deal with this problem, we must define a new form of constraint, called a *multivalued dependency*. As we did for functional dependencies, we shall use multivalued dependencies to define a normal form for relation schemas. This normal form, called **fourth normal form** (4NF), is more restrictive than BCNF. We shall see that every 4NF schema is also in BCNF but there are BCNF schemas that are not in .

Telegram: @uni_k

www.konkur.in

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Figure 7.13 Tabular representation of $\alpha \twoheadrightarrow \beta$.

7.6.1 Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $A \rightarrow B$, then we cannot have two tuples with the same A value but different B values. Multivalued dependencies, on the other hand, do not rule out the existence of certain tuples. Instead, they *require* that other tuples of a certain form be present in the relation. For this reason, functional dependencies sometimes are referred to as **equality-generating dependencies**, and multivalued dependencies are referred to as **tuple-generating dependencies**.

Let $r(R)$ be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if, in any legal instance of relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

This definition is less complicated than it appears to be. Figure 7.13 gives a tabular picture of t_1, t_2, t_3, and t_4. Intuitively, the multivalued dependency $\alpha \twoheadrightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and $R - \beta$. If the multivalued dependency $\alpha \twoheadrightarrow \beta$ is satisfied by all relations on schema R, then $\alpha \twoheadrightarrow \beta$ is a *trivial* multivalued dependency on schema R. Thus, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$. This can be seen by looking at Figure 7.13 and considering the two special cases $\beta \subseteq \alpha$ and $\beta \cup \alpha = R$. In each case, the table reduces to just two columns and we see that t_1 and t_2 are able to serve in the roles of t_3 and t_4.

To illustrate the difference between functional and multivalued dependencies, we consider the schema r_2 again, and an example relation on that schema is shown in Figure 7.14. We must repeat the department name once for each address that an instructor has, and we must repeat the address for each department with which an instructor is associated. This repetition is unnecessary, since the relationship between an instructor

Telegram: @uni_k

www.konkur.in

22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

Figure 7.14 An example of redundancy in a relation on a BCNF schema.

and his address is independent of the relationship between that instructor and a department. If an instructor with *ID* 22222 is associated with the Physics department, we want that department to be associated with all of that instructor's addresses. Thus, the relation of Figure 7.15 is illegal. To make this relation legal, we need to add the tuples (Physics, 22222, Main, Manchester) and (Math, 22222, North, Rye) to the relation of Figure 7.15.

Comparing the preceding example with our definition of multivalued dependency, we see that we want the multivalued dependency:

$$ID \twoheadrightarrow street, city$$

to hold. (The multivalued dependency $ID \twoheadrightarrow dept_name$ will do as well. We shall soon see that they are equivalent.)

As with functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies.
2. To specify constraints on the set of legal relations; we shall thus concern ourselves with *only* those relations that satisfy a given set of functional and multivalued dependencies.

Note that, if a relation r fails to satisfy a given multivalued dependency, we can construct a relation r' that *does* satisfy the multivalued dependency by adding tuples to r.

Let D denote a set of functional and multivalued dependencies. The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D. As we did for functional dependencies, we can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies. We can manage

22222	Physics	North	Rye
22222	Math	Main	Manchester

Figure 7.15 An illegal r_2 relation.

Telegram: @uni_k

www.konkur.in

with such reasoning for very simple multivalued dependencies. Luckily, multivalued dependencies that occur in practice appear to be quite simple. For complex dependencies, it is better to reason about sets of dependencies by using a system of inference rules.

From the definition of multivalued dependency, we can derive the following rules for $\alpha, \beta \subseteq R$:

- If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$. In other words, every functional dependency is also a multivalued dependency.
- If $\alpha \twoheadrightarrow \beta$, then $\alpha \twoheadrightarrow R - \alpha - \beta$

Section 28.1.1 outlines a system of inference rules for multivalued dependencies.

7.6.2 Fourth Normal Form

Consider again our example of the BCNF schema:

$$r_2 \ (ID, dept_name, street, city)$$

in which the multivalued dependency $ID \twoheadrightarrow street, city$ holds. We saw in the opening paragraphs of Section 7.6 that, although this schema is in BCNF, the design is not ideal, since we must repeat an instructor's address information for each department. We shall see that we can use the given multivalued dependency to improve the database design by decomposing this schema into a **fourth normal form** decomposition.

A relation schema R is in **fourth normal form** (**4NF**) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for R.

A database design is in 4NF if each member of the set of relation schemas that constitutes the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF in only the use of multivalued dependencies. Every 4NF schema is in BCNF. To see this fact, we note that, if a schema R is not in BCNF, then there is a nontrivial functional dependency $\alpha \rightarrow \beta$ holding on R, where α is not a superkey. Since $\alpha \rightarrow \beta$ implies $\alpha \twoheadrightarrow \beta$, R cannot be in 4NF.

Let R be a relation schema, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. To check if each relation schema R_i in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each R_i. Recall that, for a set F of functional dependencies, the restriction F_i of F to R_i is all functional dependencies in F^+ that include *only* attributes of R_i. Now consider a set D of both functional and multivalued dependencies. The **restriction** of D to R_i is the set D_i consisting of:

Telegram: @uni_k

www.konkur.in

1. All functional dependencies in D^+ that include only attributes of R_i.
2. All multivalued dependencies of the form:

$$\alpha \twoheadrightarrow \beta \cap R_i$$

where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+.

7.6.3 4NF Decomposition

The analogy between 4NF and BCNF applies to the algorithm for decomposing a schema into 4NF. Figure 7.16 shows the 4NF decomposition algorithm. It is identical to the BCNF decomposition algorithm of Figure 7.11, except that it uses multivalued dependencies and uses the restriction of D^+ to R_i.

If we apply the algorithm of Figure 7.16 to (*ID*, *dept_name*, *street*, *city*), we find that *ID*$\twoheadrightarrow$ *dept_name* is a nontrivial multivalued dependency, and *ID* is not a superkey for the schema. Following the algorithm, we replace it with two schemas:

(*ID*, *dept_name*)
(*ID*, *street*, *city*)

This pair of schemas, which is in 4NF, eliminates the redundancy we encountered earlier.

As was the case when we were dealing solely with functional dependencies, we are interested in decompositions that are lossless and that preserve dependencies. The following fact about multivalued dependencies and losslessness shows that the algorithm of Figure 7.16 generates only lossless decompositions:

---

result := {*R*};
done := false;
compute D^+; Given schema R_i, let D_i denote the restriction of D^+ to R_i
while (**not** *done*) **do**
  if (there is a schema R_i in *result* that is not in 4NF w.r.t. D_i)
    then begin
      let $\alpha \twoheadrightarrow \beta$ be a nontrivial multivalued dependency that holds
      on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \emptyset$;
      result := (*result* − R_i) ∪ (R_i − β) ∪ (α, β);
    end
  else *done* := true;

Figure 7.16 4NF decomposition algorithm.

Telegram: @uni_k

www.konkur.in

- Let $r(R)$ be a relation schema, and let D be a set of functional and multivalued dependencies on R. Let $r_1(R_1)$ and $r_2(R_2)$ form a decomposition of R. This decomposition of R is lossless if and only if at least one of the following multivalued dependencies is in D^+:

$$R_1 \cap R_2 \twoheadrightarrow R_1$$
$$R_1 \cap R_2 \twoheadrightarrow R_2$$

Recall that we stated in Section 7.2.3 that, if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$, then $r_1(R_1)$ and $r_2(R_2)$ forms a lossless decomposition of $r(R)$. The preceding fact about multivalued dependencies is a more general statement about losslessness. It says that, for *every* lossless decomposition of $r(R)$ into two schemas $r_1(R_1)$ and $r_2(R_2)$, one of the two dependencies $R_1 \cap R_2 \twoheadrightarrow R_1$ or $R_1 \cap R_2 \twoheadrightarrow R_2$ must hold. To see that this is true, we need to show first that if at least one of these dependencies holds, then $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$ and next we need to show that if $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$ then $r(R)$ must satisfy at least one of these dependencies. See the Further Reading section for references to a full proof.

The issue of dependency preservation when we decompose a relation schema becomes more complicated in the presence of multivalued dependencies. Section 28.1.2 pursues this topic.

A further complication arises from the fact that it is possible for a multivalued dependency to hold only on a proper subset of the given schema, with no way to express that multivalued dependency on that given schema. Such a multivalued dependency may appear as the result of a decomposition. Fortunately, such cases, called **embedded multivalued dependencies**, are rare. See the Further Reading section for details.

7.7 More Normal Forms

The fourth normal form is by no means the "ultimate" normal form. As we saw earlier, multivalued dependencies help us understand and eliminate some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalize multivalued dependencies and lead to another normal form called **project-join normal form (PJNF)**. PJNF is called **fifth normal form** in some books. There is a class of even more general constraints that leads to a normal form called **domain-key normal form (DKNF)**.

A practical problem with the use of these generalized constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules for reasoning about the constraints. Hence PJNF and DKNF are used quite rarely. Chapter 28 provides more details about these normal forms.

Conspicuous by its absence from our discussion of normal forms is **second normal form** (2NF). We have not discussed it because it is of historical interest only. We simply

Telegram: @uni_k

www.konkur.in

define it and let you experiment with it in Practice Exercise 7.19. First normal form deals with a different issue than the normal forms we have seen so far. It is discussed in the next section.

7.8 Atomic Domains and First Normal Form

The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. Specifically, it allows multivalued attributes such as *phone_number* in Figure 6.8 and composite attributes (such as an attribute *address* with component attributes *street*, *city*, and *state*). When we create tables from E-R designs that contain these types of attributes, we eliminate this substructure. For composite attributes, we let each component be an attribute in its own right. For multivalued attributes, we create one tuple for each item in a multivalued set.

In the relational model, we formalize this idea that attributes do not have any substructure. A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema R is in **first normal form** (**1NF**) if the domains of all attributes of R are atomic.

A set of names is an example of a non-atomic value. For example, if the schema of a relation *employee* included an attribute *children* whose domain elements are sets of names, the schema would not be in first normal form.

Composite attributes, such as an attribute *address* with component attributes *street* and *city* also have non-atomic domains.

Integers are assumed to be atomic, so the set of integers is an atomic domain; however, the set of all sets of integers is a non-atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers making up the set. But the important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be non-atomic if we considered each integer to be an ordered list of digits.

As a practical illustration of this point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be "CS001" and "EE1127". Such identification numbers can be divided into smaller units and are therefore non-atomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

When such identification numbers are used, the department of an employee can be found by writing code that breaks up the structure of an identification number. Doing so requires extra programming, and information gets encoded in the application program rather than in the database. Further problems arise if such identification numbers are used as primary keys: When an employee changes departments, the employee's identification number must be changed everywhere it occurs, which can be a difficult task, or the code that interprets the number would give a wrong result.

Telegram: @uni_k

www.konkur.in

From this discussion, it may appear that our use of course identifiers such as "CS-101", where "CS" indicates the Computer Science department, means that the domain of course identifiers is not atomic. Such a domain is not atomic as far as humans using the system are concerned. However, the database application still treats the domain as atomic, as long as it does not attempt to split the identifier and interpret parts of the identifier as a department abbreviation. The *course* schema stores the department name as a separate attribute, and the database application can use this attribute value to find the department of a course, instead of interpreting particular characters of the course identifier. Thus, our university schema can be considered to be in first normal form.

The use of set-valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies. For instance, instead of having the relationship between instructors and sections being represented as a separate relation *teaches*, a database designer may be tempted to store a set of course section identifiers with each instructor and a set of instructor identifiers with each section. (The primary keys of *section* and *instructor* are used as identifiers.) Whenever data pertaining to which instructor teaches which section is changed, the update has to be performed at two places: in the set of instructors for the section, and in the set of sections for the instructor. Failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets would avoid repeated information; however keeping only one of these would complicate some queries, and it is unclear which of the two it would be better to retain.

Some types of non-atomic values can be useful, although they should be used with care. For example, composite-valued attributes are often useful, and set-valued attributes are also useful in many cases, which is why both are supported in the E-R model. In many domains where entities have a complex structure, forcing a first normal form representation represents an unnecessary burden on the application programmer, who has to write code to convert data into atomic form. There is also the runtime overhead of converting data back and forth from the atomic form. Support for non-atomic values can thus be very useful in such domains. In fact, modern database systems do support many types of non-atomic values, as we shall see in Chapter 29 restrict ourselves to relations in first normal form, and thus all domains are atomic.

7.9 Database-Design Process

So far we have looked at detailed issues about normal forms and normalization. In this section, we study how normalization fits into the overall database-design process.

Earlier in the chapter starting in Section 7.1.1, we assumed that a relation schema $r(R)$ is given, and we proceeded to normalize it. There are several ways in which we could have come up with the schema $r(R)$:

1. $r(R)$ could have been generated in converting an E-R diagram to a set of relation schemas.

Telegram: @uni_k

www.konkur.in

2. $r(R)$ could have been a single relation schema containing *all* attributes that are of interest. The normalization process then breaks up $r(R)$ into smaller schemas.
3. $r(R)$ could have been the result of an ad hoc design of relations that we then test to verify that it satisfies a desired normal form.

In the rest of this section, we examine the implications of these approaches. We also examine some practical issues in database design, including denormalization for performance and examples of bad design that are not detected by normalization.

7.9.1 E-R Model and Normalization

When we define an E-R diagram carefully, identifying all entity sets correctly, the relation schemas generated from the E-R diagram should not need much further normalization. However, there can be functional dependencies among attributes of an entity set. For instance, suppose an *instructor* entity set had attributes *dept_name* and *dept_address*, and there is a functional dependency *dept_name* → *dept_address*. We would then need to normalize the relation generated from *instructor*.

Most examples of such dependencies arise out of poor E-R diagram design. In the preceding example, if we had designed the E-R diagram correctly, we would have created a *department* entity set with attribute *dept_address* and a relationship set between *instructor* and *department*. Similarly, a relationship set involving more than two entity sets may result in a schema that may not be in a desirable normal form. Since most relationship sets are binary, such cases are relatively rare. (In fact, some E-R-diagram variants actually make it difficult or impossible to specify nonbinary relationship sets.)

Functional dependencies can help us detect poor E-R design. If the generated relation schemas are not in desired normal form, the problem can be fixed in the E-R diagram. That is, normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling, and it can be done formally on the relation schemas generated from the E-R model.

A careful reader will have noted that in order for us to illustrate a need for multivalued dependencies and fourth normal form, we had to begin with schemas that were not derived from our E-R design. Indeed, the process of creating an E-R design tends to generate 4NF designs. If a multivalued dependency holds and is not implied by the corresponding functional dependency, it usually arises from one of the following sources:

- A many-to-many relationship set.
- A multivalued attribute of an entity set.

For a many-to-many relationship set, each related entity set has its own schema, and there is an additional schema for the relationship set. For a multivalued attribute, a separate schema is created consisting of that attribute and the primary key of the entity set (as in the case of the *phone_number* attribute of the entity set *instructor*).

Telegram: @uni_k

www.konkur.in

The universal-relation approach to relational database design starts with an assumption that there is one single relation schema containing all attributes of interest. This single schema defines how users and applications interact with the database.

7.9.2 Naming of Attributes and Relationships

A desirable feature of a database design is the **unique-role assumption**, which means that each attribute name has a unique meaning in the database. This prevents us from using the same attribute to mean different things in different schemas. For example, we might otherwise consider using the attribute *number* for phone number in the *instructor* schema and for room number in the *classroom* schema. The join of a relation on schema *instructor* with one on *classroom* is meaningless. While users and application developers can work carefully to ensure use of the right *number* in each circumstance, having a different attribute name for phone number and for room number serves to reduce user errors.

While it is a good idea to keep names for incompatible attributes distinct, if attributes of different relations have the same meaning, it may be a good idea to use the same attribute name. For this reason we used the same attribute name "*name*" for both the *instructor* and the *student* entity sets. If this was not the case (i.e., if we used different naming conventions for the instructor and student names), then if we wished to generalize these entity sets by creating a *person* entity set, we would have to rename the attribute. Thus, even if we did not currently have a generalization of *student* and *instructor*, if we foresee such a possibility, it is best to use the same name in both entity sets (and relations).

Although technically, the order of attribute names in a schema does not matter, it is a convention to list primary-key attributes first. This makes reading default output (as from **select** *) easier.

In large database schemas, relationship sets (and schemas derived therefrom) are often named via a concatenation of the names of related entity sets, perhaps with an intervening hyphen or underscore. We have used a few such names, for example, *inst_sec* and *student_sec*. We used the names *teaches* and *takes* instead of using the longer concatenated names. This was acceptable since it is not hard for you to remember the associated entity sets for a few relationship sets. We cannot always create relationship-set names by simple concatenation; for example, a manager or works-for relationship between employees would not make much sense if it were called *employee_employee*! Similarly, if there are multiple relationship sets possible between a pair of entity sets, the relationship-set names must include extra parts to identify the relationship set.

Different organizations have different conventions for naming entity sets. For example, we may call an entity set of students *student* or *students*. We have chosen to use the singular form in our database designs. Using either singular or plural is acceptable, as long as the convention is used consistently across all entity sets.

Telegram: @uni_k

www.konkur.in

As schemas grow larger, with increasing numbers of relationship sets, using consistent naming of attributes, relationships, and entities makes life much easier for the database designer and application programmers.

7.9.3 Denormalization for Performance

Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications. The penalty paid for not using a normalized schema is the extra work (in terms of coding time and execution time) to keep redundant data consistent.

For instance, suppose all course prerequisites have to be displayed along with the course information, every time a course is accessed. In our normalized schema, this requires a join of *course* with *prereq*.

One alternative to computing the join on the fly is to store a relation containing all the attributes of *course* and *prereq*. This makes displaying the "full" course information faster. However, the information for a course is repeated for every course prerequisite, and all copies must be updated by the application, whenever a course prerequisite is added or dropped. The process of taking a normalized schema and making it nonnormalized is called **denormalization**, and designers use it to tune the performance of systems to support time-critical operations.

A better alternative, supported by many database systems today, is to use the normalized schema and additionally store the join of *course* and *prereq* as a materialized view. (Recall that a materialized view is a view whose result is stored in the database and brought up to date when the relations used in the view are updated.) Like denormalization, using materialized views does have space and time overhead; however, it has the advantage that keeping the view up to date is the job of the database system, not the application programmer.

7.9.4 Other Design Issues

There are some aspects of database design that are not addressed by normalization and can thus lead to bad database design. Data pertaining to time or to ranges of time have several such issues. We give examples here; obviously, such designs should be avoided.

Consider a university database, where we want to store the total number of instructors in each department in different years. A relation *total_inst*(*dept_name*, *year*, *size*) could be used to store the desired information. The only functional dependency on this relation is *dept_name*, *year*→ *size*, and the relation is in BCNF.

An alternative design is to use multiple relations, each storing the size information for a different year. Let us say the years of interest are 2017, 2018, and 2019; we would then have relations of the form *total_inst_2017*, *total_inst_2018*, *total_inst_2019*, all of which are on the schema (*dept_name*, *size*). The only functional dependency here on each relation would be *dept_name* → *size*, so these relations are also in BCNF.

However, this alternative design is clearly a bad idea—we would have to create a new relation every year, and we would also have to write new queries every year, to take

Telegram: @uni_k

www.konkur.in

each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

Yet another way of representing the same data is to have a single relation *dept_year*(*dept_name*, *total_inst_2017*, *total_inst_2018*, *total_inst_2019*). Here the only functional dependencies are from *dept_name* to the other attributes, and again the relation is in BCNF. This design is also a bad idea since it has problems similar to the previous design—namely, we would have to modify the relation schema and write new queries every year. Queries would also be more complicated, since they may have to refer to many attributes.

Representations such as those in the *dept_year* relation, with one column for each value of an attribute, are called **crosstabs**; they are widely used in spreadsheets and reports and in data analysis tools. While such representations are useful for display to users, for the reasons just given, they are not desirable in a database design. SQL includes features to convert data from a normal relational representation to a crosstab, for display, as we discussed in Section 11.3.1.

7.10 Modeling Temporal Data

Suppose we retain data in our university organization showing not only the address of each instructor, but also all former addresses of which the university is aware. We may then ask queries, such as "Find all instructors who lived in Princeton in 1981." In this case, we may have multiple addresses for instructors. Each address has an associated start and end date, indicating when the instructor was resident at that address. A special value for the end date, for example, null, or a value well into the future, such as 9999-12-31, can be used to indicate that the instructor is still resident at that address.

In general, **temporal data** are data that have an associated time interval during which they are **valid**.[10]

Modeling temporal data is a challenging problem for several reasons. For example, suppose we have an *instructor* entity set with which we wish to associate a time-varying address. To add temporal information to an address, we would then have to create a multivalued attribute, each of whose values is a composite value containing an address and a time interval. In addition to time-varying attribute values, entities may themselves have an associated valid time. For example, a student entity may have a valid time from the date the student entered the university to the date the student graduated (or left the university). Relationships too may have associated valid times. For example, the *prereq* relationship may record when a course became a prerequisite for another course. We would thus have to add valid time intervals to attribute values, entity sets, and relationship sets. Adding such detail to an E-R diagram makes it very difficult to create and to comprehend. There have been several proposals to extend the E-R notation to

[10]There are other models of temporal data that distinguish between **valid time** and **transaction time**, the latter recording when a fact was recorded in the database. We ignore such details

Telegram: @uni_k

www.konkur.in

BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

Figure 7.17 A temporal version of the *course* relation

specify in a simple manner that an attribute value or relationship is time varying, but there are no accepted standards.

In practice, database designers fall back to simpler approaches to designing temporal databases. One commonly used approach is to design the entire database (including E-R design and relational design) ignoring temporal changes. After this, the designer studies the various relations and decides which relations require temporal variation to be tracked.

The next step is to add valid time information to each such relation by adding start and end time as attributes. For example, consider the *course* relation. The title of the course may change over time, which can be handled by adding a valid time range; the resultant schema would be:

$$course\ (course_id, title, dept_name, credits, start, end)$$

An instance of the relation is shown in Figure 7.17. Each tuple has a valid interval associated with it. Note that as per the SQL:2011 standard, the interval is **closed** on the left-hand side, that is, the tuple is valid at time *start*, but is **open** on the right-hand side, that is, the tuple is valid until just before time *end*, but is invalid at time *end*. This allows a tuple to have the same start time as the end time of another tuple, without overlapping. In general, left and right endpoints that are closed are denoted by [and], while left and right endpoints that are open are denoted by (and). Intervals in SQL:2011 are of the form [*start*, *end*), that is they are closed on the left and open on the right, Note that 9999-12-31 is the highest possible date as per the SQL standard.

It can be seen in Figure 7.17 that the title of the course CS-201 has changed several times. Suppose that on 2020-01-01 the title of the course is updated again to, say, "Intro. to Scala". Then, the *end* attribute value of the tuple with title "Intro. to Python" would be updated to 2020-01-01, and a new tuple (CS-201, Intro. to Scala, Comp. Sci., 4, 2020-01-01, 9999-12-31) would be added to the relation.

When we track data values across time, functional dependencies that we assumed to hold, such as:

$$course_id \rightarrow title, dept_name, credits$$

may no longer hold. The following constraint (expressed in English) would hold instead: "A course *course_id* has only one *title* and *dept_name* value at any given time *t*."

Telegram: @uni_k

www.konkur.in

Functional dependencies that hold at a particular point in time are called temporal functional dependencies. We use the term **snapshot** of data to mean the value of the data at a particular point in time. Thus, a snapshot of *course* data gives the values of all attributes, such as title and department, of all courses at a particular point in time. Formally, a **temporal functional dependency** $\alpha \overset{\tau}{\rightarrow} \beta$ holds on a relation schema $r(R)$ if, for all legal instances of $r(R)$, all snapshots of r satisfy the functional dependency $\alpha \rightarrow \beta$.

The original primary key for a temporal relation would no longer uniquely identify a tuple. We could try to fix the problem by adding start and end time attributes to the primary key, ensuring no two tuples have the same primary key value. However, this solution is not correct, since it is possible to store data with overlapping valid time intervals, which would not be caught by merely adding the start and end time attributes to the primary-key constraint. Instead, the temporal version of the primary key constraint must ensure that if any two tuples have the same primary key values, their valid time intervals do not overlap. Formally, if $r.A$ is a **temporal primary key** of relation r, then whenever two tuples t_1 and t_2 in r are such that $t_1.A = t_2.A$, their valid time intervals of t_1 and t_2 must not overlap.

Foreign-key constraints are also more complicated when the referenced relation is a temporal relation. A temporal foreign key should ensure that not only does each tuple in the referencing relation, say r, have a matching tuple in the referenced relation, say s, but also their time intervals are accounted for. It is not required that there be a matching tuple in s with exactly the same time interval, nor even that a single tuple in s has a time interval containing the time interval of the r tuple. Instead, we allow the time interval of the r tuple to be covered by one or more s tuples. Formally, a **temporal foreign-key** constraint from $r.A$ to $s.B$ ensures the following: for each tuple t in r, with valid time interval (l, u), there is a subset s_t of one or more tuples in s such that each tuple $s_i \in s_t$ has $s_i.B = t.A$, and further the union of the temporal intervals of all the s_i contains (l, u).

A record in a student's transcript should refer to the course title at the time when the student took the course. Thus, the referencing relation must also record time information, to identify a particular record from the *course* relation. In our university schema, *takes.course_id* is a foreign key referencing *course*. The *year* and *semester* values of a *takes* tuple could be mapped to a representative date, such as the start date of the semester; the resulting date value could be used to identify a tuple in the temporal version of the *course* relation whose valid time interval contains the specified date. Alternatively, a *takes* tuple may be associated with a valid time interval from the start date of the semester until the end date of the semester, and *course* tuples with a matching *course_id* and an overlapping valid time may be retrieved; as long as *course* tuples are not updated during a semester, there would be only one such record.

Instead of adding temporal information to each relation, some database designers create for each relation a corresponding *history* relation that stores the history of updates to the tuples. For example, a designer may leave the *course* relation unchanged,

Telegram: @uni_k

www.konkur.in

but create a relation *course_history* containing all the attributes of *course*, with an additional *timestamp* attribute indicating when a record was added to the *course_history* table. However, such a scheme has limitations, such as an inability to associate a *takes* record with the correct course title.

The SQL:2011 standard added support for temporal data. In particular, it allows existing attributes to be declared to specify a valid time interval for a tuple. For example, for the extended *course* relation we saw above, we could declare

period for *validtime* (*start, end*)

to specify that the tuple is valid in the interval specified by the *start* and *end* (which are otherwise ordinary attributes).

Temporal primary keys can be declared in SQL:2011, as illustrated below, using the extended *course* schema:

primary key (*course_id*, *validtime* **without overlaps**)

SQL:2011 also supports temporal foreign-key constraints that allow a **period** to be specified along with the referencing relation attributes, as well as with the referenced relation attributes. Most databases, with the exception of IBM DB2, Teradata, and possibly a few others, do not support temporal primary-key constraints. To the best of our knowledge, no database system currently supports temporal foreign-key constraints (Teradata allows them to be specified, but at least as of 2018, does not enforce them).

Some databases that do not directly support temporal primary-key constraints allow workarounds to enforce such constraints. For example, although PostgreSQL does not support temporal primary-key constraints natively, such constraints can be enforced using the exclude constraint feature supported by PostgreSQL. For example, consider the *course* relation, whose primary key is *course_id*. In PostgreSQL, we can add an attribute *validtime*, of type tsrange; the tsrange data type of PostgreSQL stores a timestamp range with a start and end timestamp. PostgreSQL supports an && operator on a pair of ranges, which returns true if two ranges overlap and false otherwise. The temporal primary key can be enforced by adding the following exclude constraint (a type of constraint supported by PostgreSQL) to the *course* relation as follows:

exclude (*course_id* **with** =, *validtime* **with** &&)

The above constraint ensures that if two *course* tuples have the same *course_id* value, then their *validtime* intervals do not overlap.

Relational algebra operations, such as select, project, or join, can be extended to take temporal relations as inputs and generate temporal relations as outputs. Selection and projection operations on temporal relations output tuples whose valid time intervals are the same as that of their corresponding input tuples. A **temporal join** is slightly different: the valid time of a tuple in the join result is defined as the intersection of the valid times of the tuples from which it is derived. If the valid times do not intersect, the tuple is discarded from the result. To the best of our knowledge, no database supports temporal joins natively, although they can be expressed by SQL queries that explicitly

Telegram: @uni_k

www.konkur.in

handle the temporal conditions. Predicates, such as *overlaps, contains, before, and after* and operations such as *intersection* and *difference* on pairs of intervals are supported by several database systems.

7.11 Summary

- We showed pitfalls in database design and how to design a database schema systematically in a way that avoids those pitfalls. The pitfalls included repeated information and inability to represent some information.
- Chapter 6 showed the development of a relational database design from an E-R design and when schemas may be combined safely.
- Functional dependencies are consistency constraints that are used to define two widely used normal forms, Boyce – Codd normal form (BCNF) and third normal form (3NF).
- If the decomposition is dependency preserving, all functional dependencies can be inferred logically by considering only those dependencies that apply to one relation. This permits the validity of an update to be tested without the need to compute a join of relations in the decomposition.
- A canonical cover is a set of functional dependencies equivalent to a given set of functional dependencies, that is minimized in a specific manner to eliminate extraneous attributes.
- The algorithm for decomposing relations into BCNF ensures a lossless decomposition. There are relation schemas with a given set of functional dependencies for which there is no dependency-preserving BCNF decomposition.
- A canonical cover is used to decompose a relation schema into 3NF, which is a small relaxation of the BCNF condition. This algorithm produces designs that are both lossless and dependency-preserving. Relations in 3NF may have some redundancy, but that is deemed an acceptable trade-off in cases where there is no dependency-preserving decomposition into BCNF.
- Multivalued dependencies specify certain constraints that cannot be specified with functional dependencies alone. Fourth normal form (4NF) is defined using the concept of multivalued dependencies. Section 28.1.1 gives details on reasoning about multivalued dependencies.
- Other normal forms exist, including PJNF and DKNF, which eliminate more subtle forms of redundancy. However, these are hard to work with and are rarely used. Chapter 28 gives details on these normal forms. Second normal form is of only historical interest since it provides no benefit over 3NF.
- Relational designs typically are based on simple atomic domains for each attribute. This is called first normal form.

Telegram: @uni_k

www.konkur.in

- Time plays an important role in database systems. Databases are models of the real world. Whereas most databases model the state of the real world at a point in time (at the current time), temporal databases model the states of the real world across time.
- There are possible database designs that are bad despite being lossless, dependency-preserving, and in an appropriate normal form. We showed examples of some such designs to illustrate that functional-dependency-based normalization, though highly important, is not the only aspect of good relational design.
- In order for a database to store not only current data but also historical data, the database must also store for each such tuple the time period for which the tuple is or was valid. It then becomes necessary to define temporal functional dependencies to represent the idea that the functional dependency holds at any point in time but not over the entire relation. Similarly, the join operation needs to be modified so as to appropriately join only tuples with overlapping time intervals.
- In reviewing the issues in this chapter, note that the reason we could define rigorous approaches to relational database design is that the relational data model rests on a firm mathematical foundation. That is one of the primary advantages of the relational model compared with the other data models that we have studied.

Review Terms

- Decomposition
  - Lossy decompositions
  - Lossless decompositions
- Normalization
- Functional dependencies
- Legal instance
- Superkey
- R satisfies F
- Functional dependency
  - Holds
  - Trivial
  - Trivial
- Closure of a set of functional dependencies
- Dependency preserving
- Third normal form
- Transitive dependencies
- Logically implied
- Axioms
- Armstrong's axioms
- Sound
- Complete
- Functionally determined
- Extraneous attributes
- Canonical cover
- Restriction of F to R_i
- Dependency-preserving decomposition
- Boyce – Codd normal form (BCNF)
- BCNF decomposition algorithm

Telegram: @uni_k

www.konkur.in

- Third normal form (3NF)
- 3NF decomposition algorithm
- 3NF synthesis algorithm
- Multivalued dependency
  - Equality-generating dependencies
  - Tuple-generating dependencies
  - Embedded multivalued dependencies
- Closure
- Fourth normal form (4NF)
- Restriction of D to R_i
- Fifth normal form
- Domain-key normal form (DKNF)
- Atomic domains
- First normal form (1NF)
- Unique-role assumption
- Denormalization
- Crosstabs
- Temporal data
- Snapshot
- Temporal functional dependency
- Temporal primary key
- Temporal foreign-key
- Temporal join

Practice Exercises

7.1 Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$$(A, B, C)$$
$$(A, D, E).$$

Show that this decomposition is a lossless decomposition if the following set F of functional dependencies holds:

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

7.2 List all nontrivial functional dependencies satisfied by the relation of Figure 7.18.

a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figure 7.18 Relation of Exercise 7.2.

Telegram: @uni_k

www.konkur.in

7.3 Explain how functional dependencies can be used to indicate the following:

- A one-to-one relationship set exists between entity sets *student* and *instructor*.
- A many-to-one relationship set exists between entity sets *student* and *instructor*.

7.4 Use Armstrong's axioms to prove the soundness of the union rule. (*Hint*: Use the augmentation rule to show that, if $\alpha \rightarrow \beta$, then $\alpha \rightarrow \alpha\beta$. Apply the augmentation rule again, using $\alpha \rightarrow \gamma$, and then apply the transitivity rule.)

7.5 Use Armstrong's axioms to prove the soundness of the pseudotransitivity rule.

7.6 Compute the closure of the following set F of functional dependencies for relation schema $R = (A, B, C, D, E)$.

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

List the candidate keys for R.

7.7 Using the functional dependencies of Exercise 7.6, compute the canonical cover F_c.

7.8 Consider the algorithm in Figure 7.19 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 7.8 (Section 7.4.2) and that it computes α^+ correctly.

7.9 Given the database schema $R(A, B, C)$, and a relation r on the schema R, write an SQL query to test whether the functional dependency $B \rightarrow C$ holds on relation r. Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)

7.10 Our discussion of lossless decomposition implicitly assumed that attributes on the left-hand side of a functional dependency cannot take on null values. What could go wrong on decomposition, if this property is violated?

7.11 In the BCNF decomposition algorithm, suppose you use a functional dependency $\alpha \rightarrow \beta$ to decompose a relation schema $r(\alpha, \beta, \gamma)$ into $r_1(\alpha, \beta)$ and $r_2(\alpha, \gamma)$.

a. What primary and foreign-key constraint do you expect to hold on the decomposed relations?

b. Give an example of an inconsistency that can arise due to an erroneous update, if the foreign-key constraint were not enforced on the decomposed relations above.

Telegram: @uni_k

www.konkur.in


```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
   begin
      let β → γ denote the ith FD;
      fdcount [i] := |β|;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
      appears [A] := NIL;
      for i := 1 to |F| do
         begin
            let β → γ denote the ith FD;
            if A ∈ β then add i to appears [A];
         end
   end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
   begin
      if A ∉ result then
         begin
            result := result ∪ {A};
            for each element i of appears[A] do
               begin
                  fdcount [i] := fdcount [i] − 1;
                  if fdcount [i] := 0 then
                     begin
                        let β → γ denote the ith FD;
                        addin (γ);
                     end
               end
         end
   end
```

Figure 7.19 An algorithm

Telegram: @uni_k

www.konkur.in

c. When a relation schema is decomposed into 3NF using the algorithm in Section 7.5.2, what primary and foreign-key dependencies would you expect to hold on the decomposed schema?

7.12 Let $R_1, R_2, \ldots, R_n$ be a decomposition of schema U. Let $u(U)$ be a relation, and let $r_i = \Pi_{R_i}(u)$. Show that

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

7.13 Show that the decomposition in Exercise 7.1 is not a dependency-preserving decomposition.

7.14 Show that there can be more than one canonical cover for a given set of functional dependencies, using the following set of dependencies:

$$X \rightarrow YZ,\ Y \rightarrow XZ, \text{ and } Z \rightarrow XY.$$

7.15 The algorithm to generate a canonical cover only removes one extraneous attribute at a time. Use the functional dependencies from Exercise 7.14 to show what can go wrong if two attributes inferred to be extraneous are deleted at once.

7.16 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. (*Hint*: Show that the join of all the projections onto the schemas of the decomposition cannot have more tuples than the original relation.)

7.17 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless decompositions of R' into BCNF.

7.18 Let a **prime** attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let A be an attribute that is not in α, is not in β, and for which $\beta \rightarrow A$ holds. We say that A is **transitively dependent** on α. We can restate the definition of 3NF as follows: A relation schema R is in 3NF with respect to a set F of functional dependencies if there are no nonprime attributes A in R for which A is transitively dependent on a key for R. Show that this new definition is equivalent to the original one.

7.19 A functional dependency $\alpha \rightarrow \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$; we say that β is *partially dependent* on α. A relation schema R is in **second normal form** (**2NF**) if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.

Telegram: @uni_k

www.konkur.in

- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

7.20 Give an example of a relation schema R and a set of dependencies such that R is in BCNF but is not in 4NF.

Exercises

7.21 Give a lossless decomposition into BCNF of schema R of Exercise 7.1.

7.22 Give a lossless, dependency-preserving decomposition into 3NF of schema R of Exercise 7.1.

7.23 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational-database design.

7.24 Why are certain functional dependencies called *trivial* functional dependencies?

7.25 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

7.26 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation r that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.

7.27 Use Armstrong's axioms to prove the soundness of the decomposition rule.

7.28 Using the functional dependencies of Exercise 7.6, compute B^+.

7.29 Show that the following decomposition of the schema R of Exercise 7.1 is not a lossless decomposition:

$$(A, B, C)$$
$$(C, D, E).$$

Hint: Give an example of a relation $r(R)$ such that $\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$

7.30 Consider the following set F of functional dependencies on the relation schema (A, B, C, D, E, G):

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

Telegram: @uni_k

www.konkur.in

a. Compute B^+.

b. Prove (using Armstrong's axioms) that AG is a superkey.

c. Compute a canonical cover for this set of functional dependencies F; give each step of your derivation with an explanation.

d. Give a 3NF decomposition of the given schema based on a canonical cover.

e. Give a BCNF decomposition of the given schema using the original set F of functional dependencies.

7.31 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$B \rightarrow D$$
$$DE \rightarrow B$$
$$DEG \rightarrow AB$$
$$AC \rightarrow DE$$

R is not in BCNF for many reasons, one of which arises from the functional dependency $AB \rightarrow CD$. Explain why $AB \rightarrow CD$ shows that R is not in BCNF and then use the BCNF decomposition algorithm starting with $AB \rightarrow CD$ to generate a BCNF decomposition of R. Once that is done, determine whether your result is or is not dependency preserving, and explain your reasoning.

7.32 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$A \rightarrow BC$$
$$BD \rightarrow E$$
$$CD \rightarrow AB$$

a. Find a nontrivial functional dependency containing no extraneous attributes that is logically implied by the above three dependencies and explain how you found it.

b. Use the BCNF decomposition algorithm to find a BCNF decomposition of R. Start with $A \rightarrow BC$. Explain your steps.

c. For your decomposition, state whether it is lossless and explain why.

d. For your decomposition, state whether it is dependency preserving and explain why.

Telegram: @uni_k

www.konkur.in

7.33 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$ADE \rightarrow GDE$$
$$B \rightarrow GC$$
$$G \rightarrow DE$$

Use the 3NF decomposition algorithm to generate a 3NF decomposition of R, and show your work. This means:

a. A list of all candidate keys

b. A canonical cover for F, along with an explanation of the steps you took to generate it

c. The remaining steps of the algorithm, with explanation

d. The final decomposition

7.34 Consider the schema $R = (A, B, C, D, E, G, H)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$D \rightarrow C$$
$$DE \rightarrow B$$
$$DEH \rightarrow AB$$
$$AC \rightarrow DC$$

Use the 3NF decomposition algorithm to generate a 3NF decomposition of R, and show your work. This means:

a. A list of all candidate keys

b. A canonical cover for F

c. The steps of the algorithm, with explanation

d. The final decomposition

7.35 Although the BCNF algorithm ensures that the resulting decomposition is lossless, it is possible to have a schema and a decomposition that was not generated by the algorithm, that is in BCNF, and is not lossless. Give an example of such a schema and its decomposition.

7.36 Show that every schema consisting of exactly two attributes must be in BCNF regardless of the given set F of functional dependencies.

Telegram: @uni_k

www.konkur.in

7.37 List the three design goals for relational databases, and explain why each is desirable.

7.38 In designing a relational database, why might we choose a non-BCNF design?

7.39 Given the three goals of relational database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Exercise 7.19 for the definition of 2NF.)

7.40 Given a relational schema $r(A, B, C, D)$, does $A \twoheadrightarrow BC$ logically imply $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$? If yes prove it, or else give a counter example.

7.41 Explain why 4NF is a normal form more desirable than BCNF.

7.42 Normalize the following schema, with given constraints, to 4NF.

books(*accessionno*, *isbn*, *title*, *author*, *publisher*)
users(*userid*, *name*, *deptid*, *deptname*)
accessionno $\rightarrow$ *isbn*
isbn $\rightarrow$ *title*
isbn $\rightarrow$ *publisher*
isbn $\twoheadrightarrow$ *author*
userid $\rightarrow$ *name*
userid $\rightarrow$ *deptid*
deptid $\rightarrow$ *deptname*

7.43 Although SQL does not support functional dependency constraints, if the database system supports constraints on materialized views, and materialized views are maintained immediately, it is possible to enforce functional dependency constraints in SQL. Given a relation $r(A, B, C)$, explain how constraints on materialized views can be used to enforce the functional dependency $B \rightarrow C$.

7.44 Given two relations $r(A, B, validtime)$ and $s(B, C, validtime)$, where *validtime* denotes the valid time interval, write an SQL query to compute the temporal natural join of the two relations. You can use the && operator to check if two intervals overlap and the $*$ operator to compute the intersection of two intervals.

Further Reading

The first discussion of relational database design theory appeared in an early paper by [Codd (1970)]. In that paper, Codd also introduced functional dependencies and first, second, and third normal forms.

Armstrong's axioms were introduced in [Armstrong (1974)]. BCNF was introduced in [Codd (1972)]. [Maier (1983)] is a classic textbook that provides detailed coverage of normalization and the theory of functional and multivalued dependencies.

Telegram: @uni_k

www.konkur.in

Bibliography

[Armstrong (1974)] W. W. Armstrong, "Dependency Structures of Data Base Relationships", In *Proc. of the 1974 IFIP Congress* (1974), pages 580–583.

[Codd (1970)] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Codd (1972)] E. F. Codd. "Further Normalization of the Data Base Relational Model", In *[Rustin (1972)]*, pages 33–64 (1972).

[Maier (1983)] D. Maier, *The Theory of Relational Databases*, Computer Science Press (1983).

[Rustin (1972)] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



PART 3

APPLICATION DESIGN AND DEVELOPMENT

One of the key requirements of the relational model is that data values be atomic: multivalued, composite, and other complex data types are disallowed by the core relational model. However, there are many applications where the constraints on data types imposed by the relational model cause more problems than they solve. In Chapter 8, we discuss several *complex data types*, including semistructured data types that are widely used in building applications, object-based data, textual data, and spatial data.

Practically all use of databases occurs from within application programs. Correspondingly, almost all user interaction with databases is indirect, via application programs. Database-backed applications are ubiquitous on the web as well as on mobile platforms. In Chapter 9, we study tools and technologies that are used to build applications, focusing on interactive applications that use databases to store and retrieve data.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 8



Complex Data Types

The relational model is very widely used for data representation for a large number of application domains. One of the key requirements of the relational model is that data values be atomic: multivalued, composite, and other complex data types are disallowed by the core relational model. However, there are many applications where the constraints on data types imposed by the relational model cause more problems than they solve. In this chapter, we discuss several *non-atomic data types* that are widely used, including semi-structured data, object-based data, textual data, and spatial data.

8.1 Semi-structured Data

Relational database designs have tables with a fixed number of attributes, each of which contains an atomic value. Changes to the schema, such as adding an extra attribute, are rare events, and may require changing of application code. Such a design is well suited to many organizational applications.

However, there are many application domains that need to store more complex data, whose schema changes often. Fast evolving web applications are an example of such a domain. As an example of the data management needs of such applications, consider the profile of a user which needs to be accessible to a number of different applications. The profile contains a variety of attributes, and there are frequent additions to the attributes stored in the profile. Some attributes may contain complex data; for example, an attribute may store a set of interests that can be used to show the user articles related to the set of interests. While such a set can be stored in a normalized fashion in a separate relation, a set data type allows significantly more efficient access than does a normalized representation. We study a number of data models that support representation of semi-structured data in this section.

Data exchange is another very important motivation for semi-structured data representations; it is perhaps even more important than storage for many applications. A popular architecture for building information systems today is to create a web service that allows retrieval of data and to build application code that displays the data and allows user interaction. Such application code may be developed as mobile applications,

Telegram: @uni_k

www.konkur.in

or it may be written in JavaScript and run on the browser. In either case, the ability to run on the client's machine allows developers to create very responsive user interfaces, unlike the early generation of web interfaces where backend servers send HTML marked-up text to browsers, which display the HTML. A key to building such applications is the ability to efficiently exchange and process complex data between backend servers and clients. We study the JSON and XML data models that have been widely adopted for this task.

8.1.1 Overview of Semi-structured Data Models

The relational data model has been extended in several ways to support the storage and data exchange needs of modern applications.

8.1.1.1 Flexible Schema

Some database systems allow each tuple to potentially have a different set of attributes; such a representation is referred to as a **wide column** data representation. The set of attributes is not fixed in such a representation; each tuple may have a different set of attributes, and new attributes may be added as needed.

A more restricted form of this representation is to have a fixed but very large number of attributes, with each tuple using only those attributes that it needs, leaving the rest with null values; such a representation is called a **sparse column** representation.

8.1.1.2 Multivalued Data Types

Many data representations allow attributes to contain non-atomic values. Many databases allow the storage of **sets**, **multisets**, or **arrays** as attribute values. For example, an application that stores topics of interest to a user, and uses the topics to target articles or advertisements to the user, may store the topics as a set. An example of such a set may be:

{ basketball, La Liga, cooking, anime, Jazz }

Although a set-valued attribute can be stored in a normalized form as we saw earlier in Section 6.7.2, doing so provides no benefits in this case, since lookups are always based on the user, and normalization would significantly increase the storage and querying overhead.

Some representations allow attributes to store *key-value maps*, which store key-value pairs. A **key-value map**, often just called a **map**, is a set of (*key, value*) pairs, such that each key occurs in at most one element. For example, e-commerce sites often list specifications or details for each product that they sell, such as brand, model, size, color, and numerous other product-specific details. The set of specifications may be different for each product. Such specifications can be represented as a map, where

Telegram: @uni_k

www.konkur.in

the specifications form the key, and the associated value is stored with the key. The following example illustrates such a map:

{ (brand, Apple), (ID, MacBook Air), (size, 13), (color, silver) }

The `put(key, value)` method can be used to add a key-value pair, while the `get(key)` method can be used to retrieve the value associated with a key. The `delete(key)` method can be used to delete a key-value pair from the map.

Arrays are very important for scientific and monitoring applications. For example, scientific applications may need to store images, which are basically two-dimensional arrays of pixel values. Scientific experiments as well as industrial monitoring applications often use multiple sensors that provide readings at regular intervals. Such readings can be viewed as an array. In fact, treating a stream of readings as an array requires far less space than storing each reading as a separate tuple, with attributes such as (*time*, *reading*). Not only do we avoid storing the *time* attribute explicitly (it can be inferred from the offset), but we can also reduce per-tuple overhead in the database, and most importantly we can use compression techniques to reduce the space needed to store an array of readings.

Support for multivalued attribute types was proposed early in the history of databases, and the associated data model was called the *non first-normal-form*, or *NFNF*, data model. Several relational databases such as Oracle and PostgreSQL support set and array types.

An **array database** is a database that provides specialized support for arrays, including efficient compressed storage, and query language extensions to support operations on arrays. Examples include the Oracle GeoRaster, the PostGIS extension to PostgreSQL, the SciQL extension of MonetDB, and SciDB, a database tailored for scientific applications, with a number of features tailored for array data types.

8.1.1.3 Nested Data Types

Many data representations allow attributes to be structured, directly modeling composite attributes in the E-R model. For example, an attribute *name* may have component attributes *firstname*, and *lastname*. These representations also support multivalued data types such as sets, arrays, and maps. All of these data types represent a hierarchy of data types, and that structure leads to the use of the term **nested data types**. Many databases support such types as part of their support for object-oriented data, which we describe in Section 8.2.

In this section, we outline two widely used data representations that allow values to have complex internal structures and that are flexible in that values are not forced to adhere to a fixed schema. These are the **JavaScript Object Notation** (**JSON**), which we describe in Section 8.1.2, and the **Extensible Markup Language** (**XML**), which we describe in Section 8.1.3.

Like the wide-table approach, the JSON and XML representations provide flexibility in the set of attributes that a record contains, as well as the types of these attributes.

Telegram: @uni_k

www.konkur.in

However, the JSON and XML representations permit a more flexible structuring of data, where objects could have sub-objects; each object thus corresponds to a tree structure.

Since they allow multiple pieces of information about a business object to be packaged into a single structure, the JSON and XML representations have both found significant acceptance in the context of data exchange between applications.

Today, JSON is widely used today for exchanging data between the backends and the user-facing sides of applications, such as mobile apps, and Web apps. JSON has also found favor for storing complex objects in storage systems that collect different data related to a particular user into one large object (sometimes referred to as a document), allowing data to be retrieved without the need for joins. XML is an older representation and is used by many systems for storing configuration and other information, and for data exchange.

8.1.1.4 Knowledge Representation

Representation of human knowledge has long been a goal of the artificial intelligence community. A variety of models were proposed for this task, with varying degrees of complexity; these could represent facts as well as rules about facts. With the growth of the web, a need arose to represent extremely large knowledge bases, with potentially billions of facts. The *Resource Description Format* (*RDF*) data representation is one such representation that has found very wide acceptance. The representation actually has far fewer features than earlier representations, but it was better suited to handle very large data volumes than the earlier knowledge representations.

Like the E-R model which we studied earlier, RDF models data as objects that have attributes and have relationships with other objects. RDF data can be viewed as a set of triples (3-tuples), or as a graph, with objects and attribute values modeled as nodes and relationships and attribute names as edges. We study RDF in more detail in Section 8.1.4.

8.1.2 JSON

The **JavaScript Object Notation** (**JSON**), is a textual representation of complex data types that is widely used to transmit data between applications and to store complex data. JSON supports the primitive data types integer, real and string, as well as arrays, and "objects," which are a collection of (attribute name, value) pairs.

Figure 8.1 shows an example of data represented using JSON. Since objects do not have to adhere to any fixed schema, they are basically the same as key-value maps, with the attribute names as keys and the attribute values as the associated values.

The example also illustrates arrays, shown in square brackets. In JSON, an array can be thought of as a map from integer offsets to values, with the square-bracket syntax viewed as just a convenient way of creating such maps.

JSON is today *the* primary data representation used for communication between applications and web services. Many modern applications use web services to store

Telegram: @uni_k

www.konkur.in


```
{
    "ID": "22222",
    "name": {
        "firstname: "Albert",
        "lastname: "Einstein"
    },
    "deptname": "Physics",
    "children": [
        {"firstname": "Hans", "lastname": "Einstein" },
        {"firstname": "Eduard", "lastname": "Einstein" }
    ]
}
```

Figure 8.1 Example of JSON data.

and retrieve data and to perform computations at a backend server; web services are described in more detail in Section 9.5.2. Applications invoke web services by sending parameters either as simple values such as strings or numbers, or by using JSON for more complex parameters. The web service then returns results using JSON. For example, an email user interface may invoke web services for each of these tasks: authenticating the user, fetching email header information to show a list of emails, fetching an email body, sending email, and so on.

The data exchanged in each of these steps are complex and have an internal structure. The ability of JSON to represent complex structures, and its ability to allow flexible structuring, make it a good fit for such applications.

A number of libraries are available that make it easy to transform data between the JSON representation and the object representation used in languages such as JavaScript, Java, Python, PHP, and other languages. The ease of interfacing between JSON and programming language data structures has played a significant role in the widespread use of JSON.

Unlike a relational representation, JSON is verbose and takes up more storage space for the same data. Further, parsing the text to retrieve required fields can be very CPU intensive. Compressed representations that also make it easier to retrieve values without parsing are therefore popular for storage of data. For example, a compressed binary format called BSON (short for Binary JSON) is used in many systems for storing JSON data.

The SQL language itself has been extended to support the JSON representation in several ways:

- JSON data can be stored as a JSON data type.
- SQL queries can generate JSON data from relational data:

Telegram: @uni_k

www.konkur.in

- There are SQL extensions that allow construction of JSON objects in each row of a query result. For example, PostgreSQL supports a json_build_object() function. As an example of its use, json_build_object('ID', 12345, 'name' 'Einstein') returns a JSON object {"ID": 12345, "name", "Einstein"}.
- There are also SQL extensions that allow creation of a JSON object from a collection of rows by using an aggregate function. For example, the json_agg aggregate function in PostgreSQL allows creation of a single JSON object from a collection of JSON objects. Oracle supports a similar aggregate function json_objectagg, as well as an aggregate json_arraytagg, which creates a JSON array with objects in a specified order. SQL Server supports a FOR JSON AUTO clause that formats the result of an SQL query as a JSON array, with one element per row in the SQL query.

- SQL queries can extract data from a JSON object using some form of path constructs. For example, in PostgreSQL, if a value *v* is of type JSON and has an attribute "ID", *v*–>'ID' would return the value of the "ID" attribute of *v*. Oracle supports a similar feature, using a "." instead of "–>", while SQL Server uses a function JSON_VALUE(value, path) to extract values from JSON objects using a specified path.

The exact syntax and semantics of these extensions, unfortunately, depend entirely on the specific database system. You can find references to more details on these extensions in the bibliographic notes for this chapter, available online.

8.1.3 XML

The XML data representation adds **tags** enclosed in angle brackets, <>, to mark up information in a textual representation. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the text to which the tag refers. For example, the title of a document might be marked up as follows:

```
<title>Database System Concepts</title>
```

Such tags can be used to represent relational data specifying relation names and attribute names as tags, as shown below:

```
<course>
    <course_id> CS-101 </course_id>
    <title> Intro. to Computer Science </title>
    <dept_name> Comp. Sci. </dept_name>
    <credits> 4 </credits>
</course>
```

Telegram: @uni_k

www.konkur.in


```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Route 66, Mesa Flats, Arizona 86047, USA  </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies  </name>
        <address> 1 Broadway, New York, NY, USA  </address>
    </supplier>
    <itemlist>
        <item>
            <identifier>  RS1  </identifier>
            <description> Atom powered rocket sled  </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier>  SG2  </identifier>
            <description> Superb glue  </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
        </item>
    </itemlist>
    <total_cost> 429.85 </total_cost>
    <payment_terms> Cash-on-delivery </payment_terms>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchase_order>
```

Figure 8.2 XML representation of a purchase order.

Unlike with a relational schema, new tags can be introduced easily, and with suitable names the data are "self-documenting" in that a human can understand or guess what a particular piece of data means based on the name.

Furthermore, tags can be used to create hierarchical structures, which is not possible with the relational model. Hierarchical structures are particularly important for representing business objects that must be exchanged between organizations; examples include bills, purchase orders, and so forth.

Figure 8.2, which shows how information about a purchase order can be represented in XML, illustrates a more realistic use of XML. Purchase orders are typically generated by one organization and sent to another. A purchase order contains a variety of information; the nested representation allows all information in a purchase order to

Telegram: @uni_k

www.konkur.in

be represented naturally in a single document. (Real purchase orders have considerably more information than that depicted in this simplified example.) XML provides a standard way of tagging the data; the two organizations must of course agree on what tags appear in the purchase order and what they mean.

The XQuery language was developed to support querying of XML data. Further details of XML and XQuery may be found in Chapter 30. Although XQuery implementations are available from several vendors, unlike SQL, adoption of XQuery has been relatively limited.

However, the SQL language itself has been extended to support XML in several ways:

- XML data can be stored as an XML data type.
- SQL queries can generate XML data from relational data. Such extensions are very useful for packaging related pieces of data into one XML document, which can then be sent to another application.
  The extensions allow the construction of XML representations from individual rows, as well as the creation of an XML document from a collection of rows by using an XMLAGG aggregate function.
- SQL queries can extract data from an XML data type value. For example, the XPath language supports "path expressions" that allow the extraction of desired parts of data from an XML document.

You can find more details on these extensions in Chapter 30.

8.1.4 RDF and Knowledge Graphs

The **Resource Description Framework** (**RDF**) is a data representation standard based on the entity-relationship model. We provide an overview of RDF in this section.

8.1.4.1 Triple Representation

The RDF model represents data by a set of **triples** that are in one of these two forms:

1. (*ID, attribute-name, value*)
2. (*ID1, relationship-name, ID2*)

where *ID, ID1* and *ID2* are identifiers of entities; entities are also referred to as **resources** in RDF. Note that unlike the E-R model, the RDF model only supports binary relationships, and it does not support more general n-ary relationships; we return to this issue later.

The first attribute of a triple is called its **subject**, the second attribute is called its **predicate**, and the last attribute is called its **object**. Thus, a triple has the structure (subject, predicate, object).

Telegram: @uni_k

www.konkur.in

10101	instance-of	instructor .
10101	name	"Srinivasan" .
10101	salary	"6500" .
00128	instance-of	student .
00128	name	"Zhang" .
00128	tot_cred	"102" .
comp_sci	instance-of	department .
comp_sci	dept_name	"Comp. Sci." .
biology	instance-of	department .
CS-101	instance-of	course .
CS-101	title	"Intro. to Computer Science" .
CS-101	course_dept	comp_sci .
sec1	instance-of	section .
sec1	sec_course	CS-101 .
sec1	sec_id	"1" .
sec1	semester	"Fall" .
sec1	year	"2017" .
sec1	classroom	packard-101 .
sec1	time_slot_id	"H" .
10101	inst_dept	comp_sci .
00128	stud_dept	comp_sci .
00128	takes	sec1 .
10101	teaches	sec1 .

Figure 8.3 RDF representation of part of the University database.

Figure 8.3 shows a triple representation of a small part of the University database. All attribute values are shown in quotes, while identifiers are shown without quotes. Attribute and relationship names (which form the predicate part of each triple) are also shown without quotes.

In our example, we use the ID values to identify instructors and students and *course_id* to identify courses. Each of their attributes is represented as a separate triple. The type information of objects is provided by the *instance-of* relationship; for example, 10101 is identified as an instance of instructor, while 00128 is an instance of student. To follow RDF syntax, the identifier of the Comp. Sci. department is denoted as comp_sci. Only one attribute of the department, *dept_name*, is shown. Since the primary key of *section* is composite, we have created new identifiers to identify sections; "sec1" identifies one such section, shown with its *semester*, *year* and *sec_id* attributes, and with a relationship *course* to CS-101.

Relationships shown in the figure include the *takes* and *teaches* relationships, which appear in the university schema. The departments of instructors, students and courses are shown as relationships *inst_dept*, *stud_dept* and *course_dept* respectively, following the E-R model; similarly, the classroom associated with a section is also shown as a *classroom* relationship with a classroom object (packard-101, in our example), and the

Telegram: @uni_k

www.konkur.in

course associated with a section is shown as a relationship *sec_course* between the section and the course.

As we saw, entity type information is represented using *instance-of* relationships between entities and objects representing types; type-subtype relationships can also be represented as *subtype* edges between type objects.

In contrast to the E-R model and relational schemas, RDF allows new attributes to be easily added to an object and also to create new types of relationships.

8.1.4.2 Graph Representation of RDF

The RDF representation has a very natural graph interpretation. Entities and attribute values can be considered as nodes, and attribute names and relationships can be considered as edges between the nodes. The attribute/relationship name can be viewed as the label of the corresponding edge. Figure 8.4 shows a graph representation of the data from Figure 8.3. Objects are shown as ovals, attribute values in rectangles, and relationships as edges with associated labels identifying the relationship. We have omitted the *instance-of* relationships for brevity.

A representation of information using the RDF graph model (or its variants and extensions) is referred to as a **knowledge graph**. Knowledge graphs are used for a variety of purposes. One such application is to store facts that are harvested from a variety of data sources, such as Wikipedia, Wikidata, and other sources on the web. An example of a fact is "Washington, D.C. is the capital of U.S.A." Such a fact can be represented as an edge labeled *capital-of* connecting two nodes, one representing the entity Washington, D.C., and the other representing the entity U.S.A.

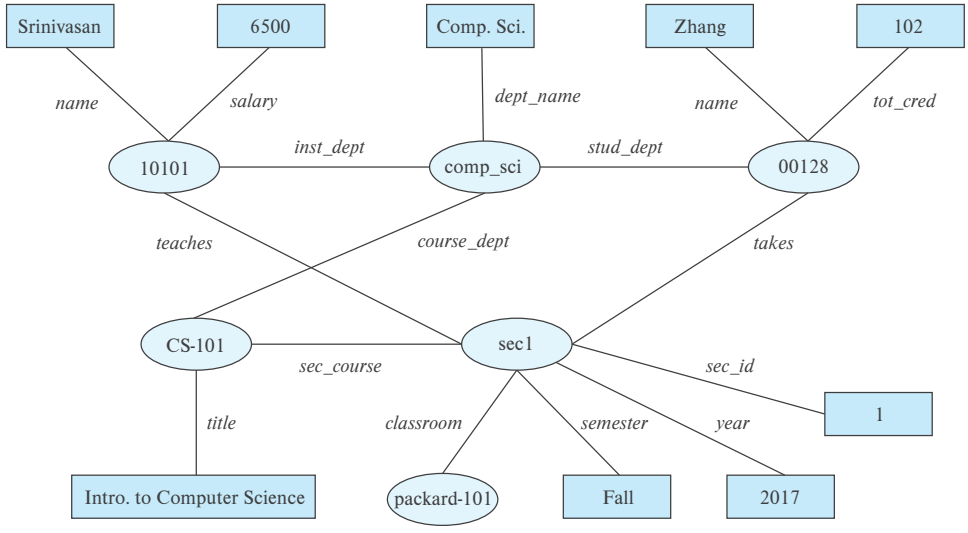


Figure 8.4 Graph representation of RDF data.

Telegram: @uni_k

www.konkur.in

Questions about entities can be answered using a knowledge graph that contains relevant information. For example, the question "Which city is the capital of the U.S.A.?" can be answered by looking for an edge labeled *capital-of*, linking an entity to the country U.S.A. (If type information is available, the query may also verify that there is an *instance-of* edge connecting Washington, D.C., to a node representing the entity type *City*).

8.1.4.3 SPARQL

SPARQL is a query language designed to query RDF data. The language is based on triple patterns, which look like RDF triples but may contain variables. For example, the triple pattern:

?cid *title* "Intro. to Computer Science"

would match all triples whose predicate is "title" and object is "Intro. to Computer Science". Here, ?cid is a variable that can match any value.

Queries can have multiple triple patterns, with variables shared across triples. Consider the following pair of triples:

?cid *title* "Intro. to Computer Science"
?sid *course* ?cid

On the university-triple dataset shown in Figure 8.3, the first triple pattern matches the triple (CS-101, title, "Intro. to Computer Science"), while the second triple pattern matches (sec1, course, CS-101). The shared variable ?cid enforces a join condition between the two triple patterns.

We can now show a complete SPARQL query. The following query retrieves names of all students who have taken a section whose course is titled "Intro. to Computer Science".

```
select ?name
where {
        ?cid title "Intro. to Computer Science" .
        ?sid course ?cid .
        ?id takes ?sid .
        ?id name ?name .
}
```

The shared variables between these triples enforce a join condition between the tuples matching each of these triples.

Note that unlike in SQL, the predicate in a triple pattern can be a variable, which can match any relationship or attribute name. has many more features, such

Telegram: @uni_k

www.konkur.in

as aggregation, optional joins (similar to outerjoins), and subqueries. For more information in SPARQL, see the references in Further Reading.

8.1.4.4 Representing N-ary Relationships

Relationships represented as edges can model only binary relationships. Knowledge graphs have been extended to store more complex relationships. For example, knowledge graphs have been extended with temporal information to record the time period during which a fact is true; if the capital of the U.S.A. changed from Washington, DC., to say, New York, in 2050, this would be represented by two facts, one for the period ending in 2050 when Washington was the capital, and one for the period after 2050.

As we saw in Section 6.9.4, an *n*-ary relationship can be represented using binary relationships by creating an artificial entity corresponding to a tuple in an *n*-ary relationship and linking that artificial entity to each of the entities participating in the relationship. In the preceding example, we can create an artificial entity e_1 to represent the fact that Barack Obama was president of the U.S.A. from 2008 to 2016. We link e_1 to the entities representing Obama and U.S.A. by *person* and *country* relationship edges respectively, and to the values 2008 and 2016 by attribute edges *president-from* and *president-till* respectively. If we chose to represent years as entities, the edges created to the two years above would represent relationships instead of attributes.

The above idea is similar to the E-R model notion of aggregation which, as we saw in Section 6.8.5, can treat a relationship as an entity; this idea is called **reification** in RDF. Reification is used in many knowledge-graph representations, where the extra information such as time period of validity are treated as *qualifiers* of the underlying edge.

Other models add a fourth attribute, called the context, to triples; thus, instead of storing triples, they store **quads**. The basic relationship is still binary, but the fourth attribute allows a context entity to be associated with a relationship. Information such as valid time period can be treated as attributes of the context entity.

There are several knowledge bases, such as Wikidata, DBPedia, Freebase, and Yago, that provide an RDF/knowledge graph representation of a wide variety of knowledge. In addition, there are a very large number of domain-specific knowledge graphs. The **linked open data** project is aimed at making a variety of such knowledge graphs open source and further creating links between these independently created knowledge graphs. Such links allow queries to make inferences using information from multiple knowledge graphs along with links to the knowledge graphs. References to more information on this topic may be found in the bibliographic notes for this chapter, available online.

8.2 Object Orientation

The **object-relational data model** extends the relational data model by providing a richer type system, including complex data types and object orientation. Relational query

Telegram: @uni_k

www.konkur.in

languages, in particular SQL, have been extended correspondingly to deal with the richer type system. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power.

Many database applications are written using an object-oriented programming language, such as Java, Python, or C++, but they need to store and fetch data from databases. Due to the type difference between the native type system of the object-oriented programming language and the relational model supported by databases, data need to be translated between the two models whenever they are fetched or stored. Merely extending the type system supported by the database was not enough to solve this problem completely. Having to express database access using a language (SQL) that is different from the programming language again makes the job of the programmer harder. It is desirable, for many applications, to have programming language constructs or extensions that permit direct access to data in the database, without having to go through an intermediate language such as SQL.

Three approaches are used in practice for integrating object orientation with database systems:

1. Build an **object-relational database system**, which adds object-oriented features to a relational database system.
2. Automatically convert data from the native object-oriented type system of the programming language to a relational representation for storage, and vice versa for retrieval. Data conversion is specified using an **object-relational mapping**.
3. Build an **object-oriented database system**, that is, a database system that natively supports an object-oriented type system and allows direct access to data from an object-oriented programming language using the native type system of the language.

We provide a brief introduction to the first two approaches in this section. While the third approach, the object-oriented database approach, has some benefits over the first two approaches in terms of language integration, it has not seen much success for two reasons. First, declarative querying is very important for efficiently accessing data, and such querying is not supported by imperative programming languages. Second, direct access to objects via pointers was found to result in increased risk of database corruption due to pointer errors. We do not describe the object-oriented approach any further.

8.2.1 Object-Relational Database Systems

In this section, we outline how object-oriented features can be added to relational database systems.

Telegram: @uni_k

www.konkur.in

8.2.1.1 User-Defined Types

Object extensions to SQL allow creation of structured user-defined types, references to such types, and tables containing tuples of such types.[1]

```
create type Person
        (ID varchar(20) primary key,
        name varchar(20),
         address varchar(20))
        ref from(ID);
create table people of Person;
```

We can create a new person as follows:

```
insert into people (ID, name, address) values
        ('12345', 'Srinivasan', '23 Coyote Run');
```

Many database systems support array and table types; attributes of relations and of user-defined types can be declared to be of such array or table types. The support for such features as well as the syntax varies widely by database system. In PostgreSQL, for example, *integer*[] denotes an array of integers whose size is not prespecified, while Oracle supports the syntax **varray**(10) **of integer** to specify an array of 10 integers. SQL Server allows table-valued types to be declared as shown in the following example:

```
create type interest as table (
        topic varchar(20),
        degree_of_interest int
);
create table users (
        ID varchar(20),
        name varchar(20),
        interests interest
);
```

User-defined types can also have methods associated with them. Only a few database systems, such as Oracle, support this feature; we omit details.

8.2.1.2 Type Inheritance

Consider the earlier definition of the type *Person* and the table *people*. We may want to store extra information in the database about people who are students and about people

[1] Structured types are different from the simpler "distinct" data types that we covered in Section 4.5.5.

Telegram: @uni_k

www.konkur.in

who are teachers. Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:

```
create type Student under Person
    (degree varchar(20)) ;
create type Teacher under Person
    (salary integer);
```

Both *Student* and *Teacher* inherit the attributes of *Person*—namely, *ID*, *name*, and *address*. *Student* and *Teacher* are said to be subtypes of *Person*, and *Person* is a supertype of *Student*, as well as of *Teacher*.

Methods of a structured type are inherited by its subtypes, just as attributes are. However, a subtype can redefine the effect of a method. We omit details.

8.2.1.3 Table Inheritance

Table inheritance allows a table to be declared as a subtable of another table and corresponds to the E-R notion of specialization/generalization. Several database systems support table inheritance, but in different ways.

In PostgreSQL, we could create a table *people* and then create tables *students* and *teachers* as subtables of *people* as follows:

```
create table students
    (degree varchar(20))
    inherits people;
create table teachers
    (salary integer)
    inherits people;
```

As a result, every attribute present in the table *people* is also present in the subtables *students* and *teachers*.

SQL:1999 supports table inheritance but requires table types to be specified first. Thus, in Oracle, which supports SQL:1999, we could use:

```
create table people of Person;
create table students of Student
    under people;
create table teachers of Teacher
    under people;
```

where the types *Student* and *Teacher* have been declared to be subtypes of *Person* as described earlier.

Telegram: @uni_k

www.konkur.in

In either case, we can insert a tuple into the *student* table as follows:

insert into *student* **values** ('00128', 'Zhang', '235 Coyote Run', 'Ph.D.');

where we provide values for the attributes inherited from *people* as well as the local attributes of *student*.

When we declare *students* and *teachers* as subtables of *people*, every tuple present in *students* or *teachers* becomes implicitly present in *people*. Thus, if a query uses the table *people*, it will find not only tuples directly inserted into that table but also tuples inserted into its subtables, namely, *students* and *teachers*. However, only those attributes that are present in *people* can be accessed by that query. SQL permits us to find tuples that are in *people* but not in its subtables by using "**only** *people*" in place of *people* in a query.

8.2.1.4 Reference Types in SQL

Some SQL implementations such as Oracle support reference types. For example, we could define the *Person* type as follows, with a reference-type declaration:

```
create type Person
        (ID varchar(20) primary key,
        name varchar(20),
         address varchar(20))
        ref from(ID);
create table people of Person;
```

By default, SQL assigns system-defined identifiers for tuples, but an existing primary-key value can be used to reference a tuple by including the **ref from** clause in the type definition as shown above.

We can define a type *Department* with a field *name* and a field *head* that is a reference to the type *Person*. We can then create a table *departments* of type *Department*, as follows:

```
create type Department (
        dept_name varchar(20),
        head ref(Person) scope people);
create table departments of Department;
```

Note that the **scope** clause above completes the definition of the foreign key from *departments.head* to the *people* relation.

When inserting a tuple for *departments*, we can then use:

```
insert into departments
        values ('CS', '12345');
```

Telegram: @uni_k

www.konkur.in

since the ID attribute is used as a reference to *Person*. Alternatively, the definition of *Person* can specify that the reference must be generated automatically by the system when a *Person* object is created. System-generated identifiers can be retrieved using **ref**(*r*) where *r* is a table name of table alias used in a query. Thus, we could create a *Person* tuple, and, using the ID or name of the person, we could retrieve the reference to the tuple in a subquery, which is used to create the value for the *head* attribute when inserting a tuple into the *departments* table. Since most database systems do not allow subqueries in an **insert into** *departments* **values** statement, the following two queries can be used to carry out the task:

```
insert into departments
    values ('CS', null);
update departments
    set head = (select ref(p)
        from people as p
        where ID = '12345')
    where dept_name = 'CS';
```

References are dereferenced in SQL:1999 by the −> symbol. Consider the *departments* table defined earlier. We can use this query to find the names and addresses of the heads of all departments:

```
select head−>name, head−>address
from departments;
```

An expression such as "*head*−>*name*" is called a **path expression**.

Since *head* is a reference to a tuple in the *people* table, the attribute *name* in the preceding query is the *name* attribute of the tuple from the *people* table. References can be used to hide join operations; in the preceding example, without the references, the *head* field of *department* would be declared a foreign key of the table *people*. To find the name and address of the head of a department, we would require an explicit join of the relations *departments* and *people*. The use of references simplifies the query considerably.

We can use the operation **deref** to return the tuple pointed to by a reference and then access its attributes, as shown below:

```
select deref(head).name
from departments;
```

8.2.2 Object-Relational Mapping

Object-relational mapping (**ORM**) systems allow a programmer to define a mapping between tuples in database relations and objects in the programming language.

Telegram: @uni_k

www.konkur.in

An object, or a set of objects, can be retrieved based on a selection condition on its attributes; relevant data are retrieved from the underlying database based on the selection conditions, and one or more objects are created from the retrieved data, based on the prespecified mapping between objects and relations.

The program can update retrieved objects, create new objects, or specify that an object is to be deleted, and then issue a save command; the mapping from objects to relations is then used to correspondingly update, insert, or delete tuples in the database.

The primary goal of object-relational mapping systems is to ease the job of programmers who build applications by providing them an object model while retaining the benefits of using a robust relational database underneath. As an added benefit, when operating on objects cached in memory, object-relational systems can provide significant performance gains over direct access to the underlying database.

Object-relational mapping systems also provide query languages that allow programmers to write queries directly on the object model; such queries are translated into SQL queries on the underlying relational database, and result objects are created from the SQL query results.

A fringe benefit of using an ORM is that any of a number of databases can be used to store data, with exactly the same high-level code. ORMs hide minor SQL differences between databases from the higher levels. Migration from one database to another is thus relatively straightforward when using an ORM, whereas SQL differences can make such migration significantly harder if an application uses SQL to communicate with the database.

On the negative side, object-relational mapping systems can suffer from significant performance inefficiencies for bulk database updates, as well as for complex queries that are written directly in the imperative language. It is possible to update the database directly, bypassing the object-relational mapping system, and to write complex queries directly in SQL in cases where such inefficiencies are discovered.

The benefits of object-relational models exceed the drawbacks for many applications, and object-relational mapping systems have seen widespread adoption in recent years. In particular, Hibernate has seen wide adoption with Java, while several ORMs including Django and SQLAlchemy are widely used with Python. More information on the Hibernate ORM system, which provides an object-relational mapping for Java, and the Django ORM system, which provides an object-relational mapping for Python, can be found in Section 9.6.2.

8.3 Textual Data

Textual data consists of unstructured text. The term **information retrieval** generally refers to the querying of unstructured textual data. In the traditional model used in the field of information retrieval, textual information is organized into *documents*. In a database, a text-valued attribute can be considered a document. In the context of the web, each web page can be considered to be a document.

Telegram: @uni_k

www.konkur.in

8.3.1 Keyword Queries

Information retrieval systems support the ability to retrieve documents with some desired information. The desired documents are typically described by a set of **keywords**—for example, the keywords "database system" may be used to locate documents on database systems, and the keywords "stock" and "scandal" may be used to locate articles about stock-market scandals. Documents have associated with them a set of keywords; typically, all the words in the documents are considered keywords. A **keyword query** retrieves documents whose set of keywords contains all the keywords in the query.

In its simplest form, an information-retrieval system locates and returns all documents that contain all the keywords in the query. More-sophisticated systems estimate the relevance of documents to a query so that the documents can be shown in order of estimated relevance. They use information about keyword occurrences, as well as hyperlink information, to estimate relevance.

Keyword search was originally targeted at document repositories within organizations or domain-specific document repositories such as research publications. But information retrieval is also important for documents stored in a database.

Keyword-based information retrieval can be used not only for retrieving textual data, but also for retrieving other types of data, such as video and audio data, that have descriptive keywords associated with them. For instance, a video movie may have associated with it keywords such as its title, director, actors, and genre, while an image or video clip may have tags, which are keywords describing the image or video clip, associated with it.

Web search engines are, at core, information retrieval systems. They retrieve and store web pages by *crawling* the web. Users submit keyword queries, and the information retrieval part of the web search engine finds stored web pages containing the required keyword. Web search engines have today evolved beyond just retrieving web pages. Today, search engines aim to satisfy a user's information needs by judging what topic a query is about and displaying not only web pages judged as relevant but also other kinds of information about the topic. For example, given a query term "cricket", a search engine may display scores from ongoing or recent cricket matches, rather than just top-ranked documents related to cricket. As another example, in response to a query "New York", a search engine may show a map of New York and images of New York in addition to web pages related to New York.

8.3.2 Relevance Ranking

The set of all documents that contain the keywords in a query may be very large; in particular, there are billions of documents on the web, and most keyword queries on a web search engine find hundreds of thousands of documents containing some or all of the keywords. Not all the documents are equally relevant to a keyword query. Information-retrieval systems therefore estimate relevance of documents to a query and return only highly ranked documents as answers. Relevance ranking is not an exact science, but there are some well-accepted approaches.

Telegram: @uni_k

www.konkur.in

8.3.2.1 Ranking Using TF-IDF

The word **term** refers to a keyword occurring in a document, or given as part of a query. The first question to address is, given a particular term t, how relevant is a particular document d to the term. One approach is to use the number of occurrences of the term in the document as a measure of its relevance, on the assumption that more relevant terms are likely to be mentioned many times in a document. Just counting the number of occurrences of a term is usually not a good indicator: first, the number of occurrences depends on the length of the document, and second, a document containing 10 occurrences of a term may not be 10 times as relevant as a document containing one occurrence.

One way of measuring $TF(d, t)$, the relevance of a term t to a document d, is:

$$TF(d, t) = \log\left(1 + \frac{n(d, t)}{n(d)}\right)$$

where $n(d)$ denotes the number of term occurrences in the document and $n(d, t)$ denotes the number of occurrences of term t in the document d. Observe that this metric takes the length of the document into account. The relevance grows with more occurrences of a term in the document, although it is not directly proportional to the number of occurrences.

Many systems refine the above metric by using other information. For instance, if the term occurs in the title, or the author list, or the abstract, the document would be considered more relevant to the term. Similarly, if the first occurrence of a term is late in the document, the document may be considered less relevant than if the first occurrence is early in the document. The above notions can be formalized by extensions of the formula we have shown for $TF(d, t)$. In the information retrieval community, the relevance of a document to a term is referred to as **term frequency** (**TF**), regardless of the exact formula used.

A query Q may contain multiple keywords. The relevance of a document to a query with two or more keywords is estimated by combining the relevance measures of the document for each keyword. A simple way of combining the measures is to add them up. However, not all terms used as keywords are equal. Suppose a query uses two terms, one of which occurs frequently, such as "database", and another that is less frequent, such as "Silberschatz". A document containing "Silberschatz" but not "database" should be ranked higher than a document containing the term "database" but not "Silberschatz".

To fix this problem, weights are assigned to terms using the **inverse document frequency** (**IDF**), defined as:

$$IDF(t) = \frac{1}{n(t)}$$

where $n(t)$ denotes the number of documents (among those indexed by the system) that contain the term t. The **relevance** of a document d to a set of terms Q is then defined as:

Telegram: @uni_k

www.konkur.in

$$r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$$

This measure can be further refined if the user is permitted to specify weights $w(t)$ for terms in the query, in which case the user-specified weights are also taken into account by multiplying $TF(t)$ by $w(t)$ in the preceding formula.

The above approach of using term frequency and inverse document frequency as a measure of the relevance of a document is called the **TF–IDF** approach.

Almost all text documents (in English) contain words such as "and," "or," "a," and so on, and hence these words are useless for querying purposes since their inverse document frequency is extremely low. Information-retrieval systems define a set of words, called **stop words**, containing 100 or so of the most common words, and ignore these words when indexing a document. Such words are not used as keywords, and they are discarded if present in the keywords supplied by the user.

Another factor taken into account when a query contains multiple terms is the **proximity** of the terms in the document. If the terms occur close to each other in the document, the document will be ranked higher than if they occur far apart. The formula for $r(d, Q)$ can be modified to take proximity of the terms into account.

Given a query Q, the job of an information-retrieval system is to return documents in descending order of their relevance to Q. Since there may be a very large number of documents that are relevant, information-retrieval systems typically return only the first few documents with the highest degree of estimated relevance and permit users to interactively request further documents.

8.3.2.2 Ranking Using Hyperlinks

Hyperlinks between documents can be used to decide on the overall importance of a document, independent of the keyword query; for example, documents linked from many other documents are considered more important.

The web search engine Google introduced **PageRank**, which is a measure of popularity of a page based on the popularity of pages that link to the page. Using the PageRank popularity measure to rank answers to a query gave results so much better than previously used ranking techniques that Google became the most widely used search engine in a rather short period of time.

Note that pages that are pointed to from many web pages are more likely to be visited, and thus should have a higher PageRank. Similarly, pages pointed to by web pages with a high PageRank will also have a higher probability of being visited, and thus should have a higher PageRank.

The PageRank of a document d is thus defined (circularly) based on the PageRank of other documents that link to document d. PageRank can be defined by a set of linear equations, as follows: First, web pages are given integer identifiers. The jump probability matrix T is defined with $T[i, j]$ set to the probability that a random walker who is following a link out of page i follows the link to page j. Assuming that each link

Telegram: @uni_k

www.konkur.in

from i has an equal probability of being followed $T[i,j] = 1/N_i$, where N_i is the number of links out of page i. Then the PageRank $P[j]$ for each page j can be defined as:

$$P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^{N} (T[i,j] * P[i])$$

where δ is a constant between 0 and 1, usually set to 0.15, and N is the number of pages.

The set of equations generated as above are usually solved by an iterative technique, starting with each $P[i]$ set to $1/N$. Each step of the iteration computes new values for each $P[i]$ using the P values from the previous iteration. Iteration stops when the maximum change in any $P[i]$ value in an iteration goes below some cutoff value.

Note that PageRank is a static measure, independent of the keyword query; given a keyword query, it is used in combination with TF-IDF scores of a document to judge its relevance of the document to the keyword query.

PageRank is not the only measure of the popularity of a site. Information about how often a site is visited is another useful measure of popularity. Further, search engines track what fraction of times users click on a page when it is returned as an answer. Keywords that occur in the anchor text associated with the hyperlink to a page are viewed as very important and are given a higher term frequency. These and a number of other factors are used to rank answers to a keyword query.

8.3.3 Measuring Retrieval Effectiveness

Ranking of results of a keyword query is not an exact science. Two metrics are used to measure how well an information-retrieval system is able to answer queries. The first, **precision**, measures what percentage of the retrieved documents are actually relevant to the query. The second, **recall**, measures what percentage of the documents relevant to the query were retrieved. Since search engines find a very large number of answers, and users typically stop after browsing some number (say, 10 or 20) of the answers, the precision and recall numbers are usually measured "@K", where K is the number of answers viewed. Thus, one can talk of precision@10 or recall@20.

8.3.4 Keyword Querying on Structured Data and Knowledge Graphs

Although querying on structured data are typically done using query languages such as SQL, users who are not familiar with the schema or the query language find it difficult to get information from such data. Based on the success of keyword querying in the context of information retrieval from the web, techniques have been developed to support keyword queries on structured and semi-structured data.

One approach is to represent the data using graphs, and then perform keyword queries on the graphs. For example, tuples can be treated as nodes in the graph, and foreign key and other connections between tuples can be treated as edges in the graph. Keyword search is then modeled as finding tuples containing the given keywords and finding connecting paths between them in the corresponding graph.

Telegram: @uni_k

www.konkur.in

For example, a query "Zhang Katz" on a university database may find the *name* "Zhang" occurring in a *student* tuple, and the *name* "Katz" in an *instructor* tuple, a path through the *advisor* relation connecting the two tuples. Other paths, such as student "Zhang" taking a course taught by "Katz" may also be found in response to this query. Such queries may be used for ad hoc browsing and querying of data when the user does not know the exact schema and does not wish to take the effort to write an SQL query defining what she is searching for. Indeed it is unreasonable to expect lay users to write queries in a structured query language, whereas keyword querying is quite natural.

Since queries are not fully defined, they may have many different types of answers, which must be ranked. A number of techniques have been proposed to rank answers in such a setting, based on the lengths of connecting paths and on techniques for assigning directions and weights to edges. Techniques have also been proposed for assigning popularity ranks to tuples based on foreign key links. More information on keyword searching of structured data may be found in the bibliographic notes for this chapter, available online.

Further, knowledge graphs can be used along with textual information to answer queries. For example, knowledge graphs can be used to provide unique identifiers to entities, which are used to annotate mentions of the entities in textual documents. Now a particular mention of a person in a document may have the phrase "Stonebraker developed PostgreSQL"; from the context, the word Stonebraker may be inferred to be the database researcher "Michael Stonebraker" and annotated by linking the word Stonebraker to the entity "Michael Stonebraker". The knowledge graph may also record the fact that Stonebraker won the Turing award. A query asking for "turing award postgresql" can now be answered by using information from the document and the knowledge graph.[2]

Web search engines today use large knowledge graphs, in addition to crawled documents, to answer user queries.

8.4 Spatial Data

Spatial data support in database systems is important for efficiently storing, indexing, and querying of data on the basis of spatial locations.

Two types of spatial data are particularly important:

- **Geographic data** such as road maps, land-usage maps, topographic elevation maps, political maps showing boundaries, land-ownership maps, and so on. **Geographic information systems** are special-purpose database systems tailored for storing geographic data. Geographic data is based on a round-earth coordinate system, with latitude, longitude, and elevation.

[2]In this case the knowledge graph may already record that Stonebraker developed PostgreSQL, but there are many other pieces of information that may exist only in documents, and

Telegram: @uni_k

www.konkur.in

- **Geometric data**, which include spatial information about how objects—such as buildings, cars, or aircraft—are constructed. Geometric data is based on a two-dimensional or three-dimensional Euclidean space, with X, Y, and Z coordinates.

Geographic and geometric data types are supported by many database systems, such as Oracle Spatial and Graph, the PostGIS extension of PostgreSQL, SQL Server, and the IBM DB2 Spatial Extender.

In this section we describe the modeling and querying of spatial data; implementation techniques such as indexing and query processing techniques are covered in Chapter 14 and in Chapter 15.

The syntax for representing geographic and geometric data varies by database, although representations based on the **Open Geospatial Consortium** (**OGC**) standard are now increasingly supported. See the manuals of the database you use to learn more about the specific syntax supported by the database.

8.4.1 Representation of Geometric Information

Figure 8.5 illustrates how various geometric constructs can be represented in a database, in a normalized fashion. We stress here that geometric information can be represented in several different ways, only some of which we describe.

A *line segment* can be represented by the coordinates of its endpoints. For example, in a map database, the two coordinates of a point would be its latitude and longitude. A **polyline** (also called a **linestring**) consists of a connected sequence of line segments and can be represented by a list containing the coordinates of the endpoints of the segments, in sequence. We can approximately represent an arbitrary curve with polylines by partitioning the curve into a sequence of segments. This representation is useful for two-dimensional features such as roads; here, the width of the road is small enough relative to the size of the full map that it can be considered to be a line. Some systems also support *circular arcs* as primitives, allowing curves to be represented as sequences of arcs.

We can represent a *polygon* by listing its vertices in order, as in Figure 8.5.[3] The list of vertices specifies the boundary of a polygonal region. In an alternative representation, a polygon can be divided into a set of triangles, as shown in Figure 8.5. This process is called **triangulation**, and any polygon can be triangulated. The complex polygon can be given an identifier, and each of the triangles into which it is divided carries the identifier of the polygon. Circles and ellipses can be represented by corresponding types or approximated by polygons.

List-based representations of polylines or polygons are often convenient for query processing. Such non-first-normal-form representations are used when supported by the underlying database. So that we can use fixed-size tuples (in first normal form) for representing polylines, we can give the polyline or curve an identifier, and we can

[3]Some references use the term *closed polygon* to refer to what we call

Telegram: @uni_k

www.konkur.in

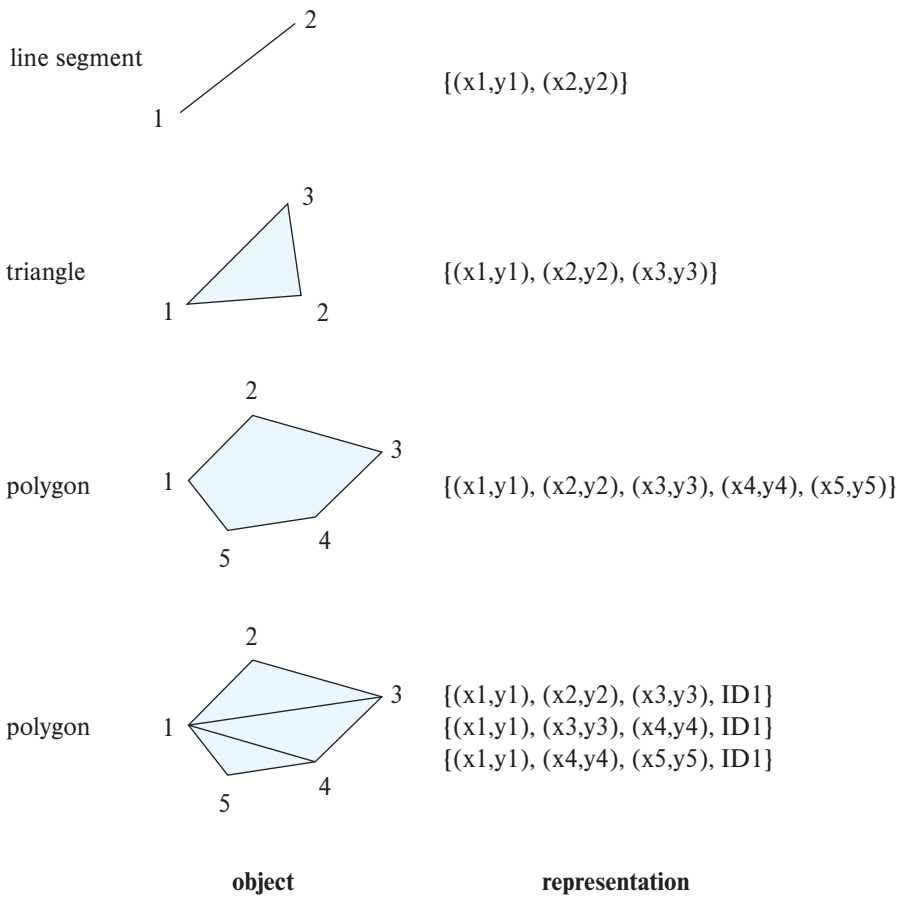


Figure 8.5 Representation of geometric constructs.

represent each segment as a separate tuple that also carries with it the identifier of the polyline or curve. Similarly, the triangulated representation of polygons allows a first normal form relational representation of polygons.

The representation of points and line segments in three-dimensional space is similar to their representation in two-dimensional space, the only difference being that points have an extra z component. Similarly, the representation of planar figures—such as triangles, rectangles, and other polygons—does not change much when we move to three dimensions. Tetrahedrons and cuboids can be represented in the same way as triangles and rectangles. We can represent arbitrary polyhedra by dividing them into tetrahedrons, just as we triangulate polygons. We can also represent them by listing their faces, each of which is itself a polygon, along with an indication of which side of the face is inside the polyhedron.

Telegram: @uni_k

www.konkur.in

For example, SQL Server and PostGIS support the **geometry** and **geography** types, each of which has subtypes such as point, linestring, curve, polygon, as well as collections of these types called multipoint, multilinestring, multicurve and multipolygon. Textual representations of these types are defined by the OGC standards, and can be converted to internal representations using conversion functions. For example, LINESTRING(1 1, 2 3, 4 4) defines a line that connects points (1, 1), (2, 3) and (4, 4), while POLYGON((1 1, 2 3, 4 4, 1 1)) defines a triangle defined by these points. Functions *ST_GeometryFromText*() and *ST_GeographyFromText*() convert the textual representations to geometry and geography objects respectively. Operations on geometry and geography types that return objects of the same type include the *ST_Union*() and *ST_Intersection*() functions which compute the union and intersection of geometric objects such as linestrings and polygons. The function names as well as syntax differ by system; see the system manuals for details.

In the context of map data, the various line segments representing the roads are actually interconnected to form a graph. Such a **spatial network** or **spatial graph** has spatial locations for vertices of the graph, along with interconnection information between the vertices, which form the edges of the graph. The edges have a variety of associated information, such as distance, number of lanes, average speed at different times of the day, and so on.

8.4.2 Design Databases

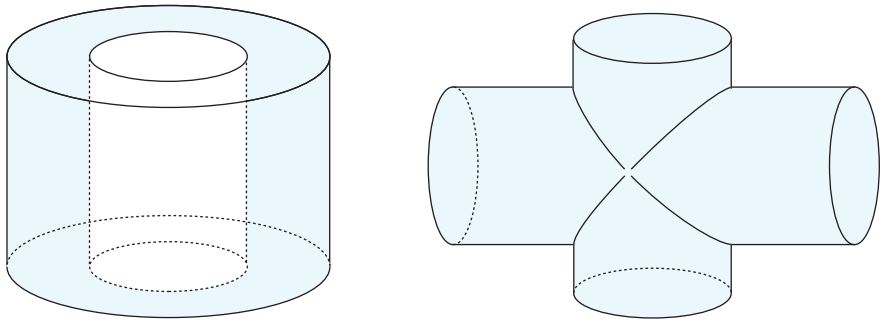
Computer-aided-design (CAD) systems traditionally stored data in memory during editing or other processing and wrote the data back to a file at the end of a session of editing. The drawbacks of such a scheme include the cost (programming complexity, as well as time cost) of transforming data from one form to another and the need to read in an entire file even if only parts of it are required. For large designs, such as the design of a large-scale integrated circuit or the design of an entire airplane, it may be impossible to hold the complete design in memory. Designers of object-oriented databases were motivated in large part by the database requirements of CAD systems. Object-oriented databases represent components of the design as objects, and the connections between the objects indicate how the design is structured.

The objects stored in a design database are generally geometric objects. Simple two-dimensional geometric objects include points, lines, triangles, rectangles, and, in general, polygons. Complex two-dimensional objects can be formed from simple objects by means of union, intersection, and difference operations. Similarly, complex three-dimensional objects may be formed from simpler objects such as spheres, cylinders, and cuboids by union, intersection, and difference operations, as in Figure 8.6. Three-dimensional surfaces may also be represented by **wireframe models**, which essentially model the surface as a set of simpler objects, such as line segments, triangles, and rectangles.

Design databases also store nonspatial information about objects, such as the material from which the objects are constructed. We can usually model such information

Telegram: @uni_k

www.konkur.in



(a) Difference of cylinders (b) Union of cylinders

Figure 8.6 Complex three-dimensional objects.

by standard data-modeling techniques. We concern ourselves here with only the spatial aspects.

Various spatial operations must be performed on a design. For instance, the designer may want to retrieve that part of the design that corresponds to a particular region of interest. Spatial-index structures, discussed in Section 14.10.1, are useful for such tasks. Spatial-index structures are multidimensional, dealing with two- and three-dimensional data, rather than dealing with just the simple one-dimensional ordering provided by the B^+-trees.

Spatial-integrity constraints, such as "two pipes should not be in the same location," are important in design databases to prevent interference errors. Such errors often occur if the design is performed manually and are detected only when a prototype is being constructed. As a result, these errors can be expensive to fix. Database support for spatial-integrity constraints helps people to avoid design errors, thereby keeping the design consistent. Implementing such integrity checks again depends on the availability of efficient multidimensional index structures.

8.4.3 Geographic Data

Geographic data are spatial in nature but differ from design data in certain ways. Maps and satellite images are typical examples of geographic data. Maps may provide not only location information—about boundaries, rivers, and roads, for example—but also much more detailed information associated with locations, such as elevation, soil type, land usage, and annual rainfall.

8.4.3.1 Applications of Geographic Data

Geographic databases have a variety of uses, including online map and navigation services, which are ubiquitous today. Other applications include distribution-network information for public-service utilities such as telephone, electric-power, and water-supply

Telegram: @uni_k

www.konkur.in

systems, and land-usage information for ecologists and planners, land records to track land ownership, and many more.

Geographic databases for public-utility information have become very important as the network of buried cables and pipes has grown. Without detailed maps, work carried out by one utility may damage the structure of another utility, resulting in large-scale disruption of service. Geographic databases, coupled with accurate location-finding systems using GPS help avoid such problems.

8.4.3.2 Representation of Geographic Data

Geographic data can be categorized into two types:

- **Raster data**. Such data consist of bitmaps or pixel maps, in two or more dimensions. A typical example of a two-dimensional raster image is a satellite image of an area. In addition to the actual image, the data include the location of the image, specified, for example, by the latitude and longitude of its corners, and the resolution, specified either by the total number of pixels, or, more commonly in the context of geographic data, by the area covered by each pixel.

  Raster data are often represented as **tiles**, each covering a fixed-size area. A larger area can be displayed by displaying all the tiles that overlap with the area. To allow the display of data at different zoom levels, a separate set of tiles is created for each zoom level. Once the zoom level is set by the user interface (e.g., a web browser), tiles at the specified zoom level that overlap the area being displayed are retrieved and displayed.

  Raster data can be three-dimensional—for example, the temperature at different altitudes at different regions, again measured with the help of a satellite. Time could form another dimension—for example, the surface temperature measurements at different points in time.

- **Vector data**. Vector data are constructed from basic geometric objects, such as points, line segments, polylines, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions. In the context of geographic data, points are usually represented by latitude and longitude, and where the height is relevant, additionally by elevation.

  Map data are often represented in vector format. Roads are often represented as polylines. Geographic features, such as large lakes, or even political features such as states and countries, are represented as complex polygons. Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant.

Geographic information related to regions, such as annual rainfall, can be represented as an array—that is, in raster form. For space efficiency, the array can be stored in a compressed form. In Section 24.4.1, we study an alternative representation of such arrays by a data structure called a *quadtree*.

Telegram: @uni_k

www.konkur.in

As another alternative, we can represent region information in vector form, using polygons, where each polygon is a region within which the array value is the same. The vector representation is more compact than the raster representation in some applications. It is also more accurate for some tasks, such as depicting roads, where dividing the region into pixels (which may be fairly large) leads to a loss of precision in location information. However, the vector representation is unsuitable for applications where the data are intrinsically raster based, such as satellite images.

Topographical information, that is information about the elevation (height) of each point on a surface, can be represented in raster form. Alternatively, it can be represented in vector form by dividing the surface into polygons covering regions of (approximately) equal elevation, with a single elevation value associated with each polygon. As another alternative, the surface can be **triangulated** (i.e., divided into triangles), with each triangle represented by the latitude, longitude, and elevation of each of its corners. The latter representation, called the **triangulated irregular network** (**TIN**) representation, is a compact representation which is particularly useful for generating three-dimensional views of an area.

Geographic information systems usually contain both raster and vector data, and they can merge the two kinds of data when displaying results to users. For example, map applications usually contain both satellite images and vector data about roads, buildings, and other landmarks. A map display usually **overlays** different kinds of information; for example, road information can be overlaid on a background satellite image to create a hybrid display. In fact, a map typically consists of multiple layers, which are displayed in bottom-to-top order; data from higher layers appear on top of data from lower layers.

It is also interesting to note that even information that is actually stored in vector form may be converted to raster form before it is sent to a user interface such as a web browser. One reason is that even web browsers in which JavaScript has been disabled can then display map data; a second reason may be to prevent end users from extracting and using the vector data.

Map services such as Google Maps and Bing Maps provide APIs that allow users to create specialized map displays, containing application-specific data overlaid on top of standard map data. For example, a web site may show a map of an area with information about restaurants overlaid on the map. The overlays can be constructed dynamically, displaying only restaurants with a specific cuisine, for example, or allowing users to change the zoom level or pan the display.

8.4.4 Spatial Queries

There are a number of types of queries that involve spatial locations.

- **Region queries** deal with spatial regions. Such a query can ask for objects that lie partially or fully inside a specified region. A query to find all retail shops within the geographic boundaries of a given town is an example. PostGIS supports predicates between two geometry or geography objects such as *ST_Contains*(), *ST_Overlaps*(),

Telegram: @uni_k

www.konkur.in

ST_Disjoint() and *ST_Touches*(). These can be used to find objects that are contained in, or intersect, or are disjoint from a region. SQL Server supports equivalent functions with slightly different names.
Suppose we have a *shop* relation, with an attribute *location* of type *point*, and a geography object of type *polygon*. Then the *ST_Contains*() function can be used to retrieve all shops whose location is contained in the given polygon.

- **Nearness queries** request objects that lie near a specified location. A query to find all restaurants that lie within a given distance of a given point is an example of a nearness query. The **nearest-neighbor query** requests the object that is nearest to a specified point. For example, we may want to find the nearest gasoline station. Note that this query does not have to specify a limit on the distance, and hence we can ask it even if we have no idea how far the nearest gasoline station lies.
  The PostGIS *ST_Distance*() function gives the minimum distance between two such objects, and can be used to find objects that are within a specified distance from a point or region. Nearest neighbors can be found by finding objects with minimum distance.
- **Spatial graph queries** request information based on spatial graphs such as road maps. For example, a query may ask for the shortest path between two locations via the road network, or via a train network, each of which can be represented as a spatial graph. Such queries are ubiquitous for navigation systems.

Queries that compute intersections of regions can be thought of as computing the **spatial join** of two spatial relations—for example, one representing rainfall and the other representing population density—with the location playing the role of join attribute. In general, given two relations, each containing spatial objects, the spatial join of the two relations generates either pairs of objects that intersect or the intersection regions of such pairs. Spatial predicates such as *ST_Contains*() or *ST_Overlaps*() can be used as join predicates when performing spatial joins.

In general, queries on spatial data may have a combination of spatial and nonspatial requirements. For instance, we may want to find the nearest restaurant that has vegetarian selections and that charges less than $10 for a meal.

8.5 Summary

- There are many application domains that need to store more complex data than simple tables with a fixed number of attributes.
- The SQL standard includes extensions of the SQL data-definition and query language to deal with new data types and with object orientation. These include support for collection-valued attributes, inheritance, and tuple references. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power.

Telegram: @uni_k

www.konkur.in

- Semi-structured data are characterized by complex data, whose schema changes often.
- A popular architecture for building information systems today is to create a web service that allows retrieval of data and to build application code that displays the data and allows user interaction.
- The relational data model has been extended in several ways to support the storage and data exchange needs of modern applications.
  - Some database systems allow each tuple to potentially have a different set of attributes.
  - Many data representations allow attributes to non-atomic values.
  - Many data representations allow attributes to be structured, directly modeling composite attributes in the E-R model.
- The **JavaScript Object Notation** (JSON) is a textual representation of complex data types which is widely used for transmitting data between applications and for storing complex data.
- XML representations provide flexibility in the set of attributes that a record contains as well as the types of these attributes.
- The **Resource Description Framework** (RDF) is a data representation standard based on the entity-relationship model. The RDF representation has a very natural graph interpretation. Entities and attribute values can be considered nodes, and attribute names and relationships can be considered edges between the nodes.
- SPARQL is a query language designed to query RDF data and is based on triple patterns.
- Object orientation provides inheritance with subtypes and subtables as well as object (tuple) references.
- The object-relational data model extends the relational data model by providing a richer type system, including collection types and object orientation.
- Object-relational database systems (i.e., database systems based on the object-relational model) provide a convenient migration path for users of relational databases who wish to use object-oriented features.
- Object-relational mapping systems provide an object view of data that are stored in a relational database. Objects are transient, and there is no notion of persistent object identity. Objects are created on demand from relational data, and updates to objects are implemented by updating the relational data. Object-relational mapping systems have been widely adopted, unlike the more limited adoption of persistent programming languages.

Telegram: @uni_k

www.konkur.in

- Information-retrieval systems are used to store and query textual data such as documents. They use a simpler data model than do database systems but provide more powerful querying capabilities within the restricted model.
- Queries attempt to locate documents that are of interest by specifying, for example, sets of keywords. The query that a user has in mind usually cannot be stated precisely; hence, information-retrieval systems order answers on the basis of potential relevance.
- Relevance ranking makes use of several types of information, such as:
  - Term frequency: how important each term is to each document.
  - Inverse document frequency.
  - Popularity ranking.
- Spatial data management is important for many applications. Geometric and geographic data types are supported by many database systems, with subtypes including points, linestrings and polygons. Region queries, nearest neighbor queries, and spatial graph queries are among the commonly used types of spatial queries.

Review Terms

- Wide column
- Sparse column
- Key-value map
- Map
- Array database
- Tags
- Triples
- Resources
- Subject
- Predicate
- Object
- Knowledge graph
- Reification
- Quads
- Linked open data
- Object-relational data model
- Object-relational database system
- Object-relational mapping
- Object-oriented database system
- Path expression
- Keywords
- Keyword query
- Term
- Relevance
- TF-IDF
- Stop words
- Proximity
- PageRank
- Precision
- Recall
- Geographic data
- Geometric data
- Geographic information system
- Computer-aided-design (CAD)
- Polyline
- Linestring

Telegram: @uni_k

www.konkur.in

- Triangulation
- Spatial network
- Spatial graph
- Raster data
- Tiles
- Vector data
- Topographical information
- Triangulated irregular network (TIN)
- Overlays
- Nearness queries
- Nearest-neighbor query
- Region queries
- Spatial graph queries
- Spatial join

Practice Exercises

8.1 Provide information about the student named Shankar in our sample university database, including information from the *student* tuple corresponding to Shankar, the *takes* tuples corresponding to Shankar and the *course* tuples corresponding to these *takes* tuples, in each of the following representations:

a. Using JSON, with an appropriate nested representation.

b. Using XML, with the same nested representation.

c. Using RDF triples.

d. As an RDF graph.

8.2 Consider the RDF representation of information from the university schema as shown in Figure 8.3. Write the following queries in SPARQL.

a. Find the titles of all courses taken by any student named Zhang.

b. Find titles of all courses such that a student named Zhang takes a section of the course that is taught by an instructor named Srinivasan.

c. Find the attribute names and values of all attributes of the instructor named Srinivasan, without enumerating the attribute names in your query.

8.3 A car-rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

- Trucks: cargo capacity.
- Sports cars: horsepower, renter age requirement.
- Vans: number of passengers.
- Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive).

Telegram: @uni_k

www.konkur.in

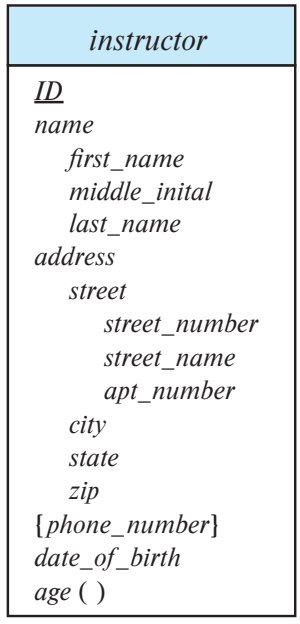


Figure 8.7 E-R diagram with composite, multivalued, and derived attributes.

Construct an SQL schema definition for this database. Use inheritance where appropriate.

8.4 Consider a database schema with a relation *Emp* whose attributes are as shown below, with types specified for multivalued attributes.

Emp = (ename, ChildrenSet **multiset***(Children), SkillSet* **multiset***(Skills))*
Children = (name, birthday)
Skills = (type, ExamSet **setof***(Exams))*
Exams = (year, city)

Define the above schema in SQL, using the SQL Server table type syntax from Section 8.2.1.1 to declare multiset attributes.

8.5 Consider the E-R diagram in Figure 8.7 showing entity set *instructor*. Give an SQL schema definition corresponding to the E-R diagram, treating *phone_number* as an array of 10 elements, using Oracle or PostgreSQL syntax.

8.6 Consider the relational schema shown in Figure 8.8.

a. Give a schema definition in SQL corresponding to the relational schema but using references to express foreign-key relationships.

b. Write each of the following queries on the schema, using SQL.

i. Find the company with the most employees.

Telegram: @uni_k

www.konkur.in

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*person_name*, *manager_name*)

Figure 8.8 Relational database for Exercise 8.6.

ii. Find the company with the smallest payroll.

iii. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

8.7 Compute the relevance (using appropriate definitions of term frequency and inverse document frequency) of each of the Practice Exercises in this chapter to the query "SQL relation".

8.8 Show how to represent the matrices used for computing PageRank as relations. Then write an SQL query that implements one iterative step of the iterative technique for finding PageRank; the entire algorithm can then be implemented as a loop containing the query.

8.9 Suppose the *student* relation has an attribute named *location* of type point, and the *classroom* relation has an attribute *location* of type polygon. Write the following queries in SQL using the PostGIS spatial functions and predicates that we saw earlier:

a. Find the names of all students whose location is within the classroom Packard 101.

b. Find all classrooms that are within 100 meters or Packard 101; assume all distances are represented in units of meters.

c. Find the ID and name of student who is geographically nearest to the student with ID 12345.

d. Find the ID and names of all pairs of students whose locations are less than 200 meters apart.

Exercises

8.10 Redesign the database of Exercise 8.4 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first and fourth normal form schemas.

Telegram: @uni_k

www.konkur.in

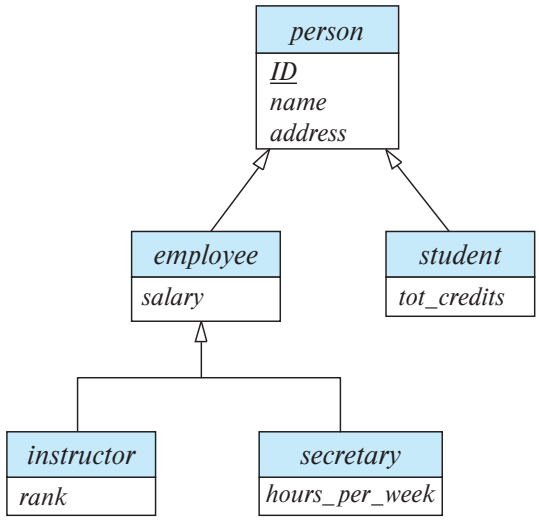


Figure 8.9 Specialization and generalization.

8.11 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 8.2.1.3. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.

8.12 Consider the E-R diagram in Figure 8.9, which contains specializations, using subtypes and subtables.

a. Give an SQL schema definition of the E-R diagram.

b. Give an SQL query to find the names of all people who are not secretaries.

c. Give an SQL query to print the names of people who are neither employees nor students.

d. Can you create a person who is an employee and a student with the schema you created? Explain how, or explain why it is not possible.

8.13 Suppose you wish to perform keyword querying on a set of tuples in a database, where each tuple has only a few attributes, each containing only a few words. Does the concept of term frequency make sense in this context? And that of inverse document frequency? Explain your answer. Also suggest how you can define the similarity of two tuples using TF-IDF concepts.

8.14 Web sites that want to get some publicity can join a web ring, where they create links to other sites in the ring in exchange for other sites in the ring creating links

Telegram: @uni_k

www.konkur.in

to their site. What is the effect of such rings on popularity ranking techniques such as PageRank?

8.15 The Google search engine provides a feature whereby web sites can display advertisements supplied by Google. The advertisements supplied are based on the contents of the page. Suggest how Google might choose which advertisements to supply for a page, given the page contents.

Further Reading

A tutorial on JSON can be found at www.w3schools.com/js/js_json_intro.asp. More information about XML can be found in Chapter 30, available online. More information about RDF can be found at www.w3.org/RDF/. Apache Jena provides an RDF implementation, with support for SPARQL; a tutorial on SPARQL can be found at jena.apache.org/tutorials/sparql.html

POSTGRES ([Stonebraker and Rowe (1986)] and [Stonebraker (1986)]) was an early implementation of an object-relational system. Oracle provides a fairly complete implementation of the object-relational features of SQL, while PostgreSQL provides a smaller subset of those features. More information on support for these features may be found in their respective manuals.

[Salton (1989)] is an early textbook on information-retrieval systems, while [Manning et al. (2008)] is a modern textbook on the subject. Information about spatial database support in Oracle, PostgreSQL and SQL Server may be found in their respective manuals online.

Bibliography

[Manning et al. (2008)] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press (2008).

[Salton (1989)] G. Salton, *Automatic Text Processing*, Addison Wesley (1989).

[Stonebraker (1986)] M. Stonebraker, "Inclusion of New Types in Relational Database Systems", In *Proc. of the International Conf. on Data Engineering* (1986), pages 262–269.

[Stonebraker and Rowe (1986)] M. Stonebraker and L. Rowe, "The Design of POSTGRES", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986), pages 340–355.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 9



Application Development

Practically all use of databases occurs from within application programs. Correspondingly, almost all user interaction with databases is indirect, via application programs. In this chapter, we study tools and technologies that are used to build applications, focusing on interactive applications that use databases to store and retrieve data.

A key requirement for any user-centric application is a good user interface. The two most common types of user interfaces today for database-backed applications are the web and mobile app interfaces.

In the initial part of this chapter, we provide an introduction to application programs and user interfaces (Section 9.1), and to web technologies (Section 9.2). We then discuss development of web applications using the widely used Java Servlets technology at the back end (Section 9.3), and using other frameworks (Section 9.4). Client-side code implemented using JavaScript or mobile app technologies is crucial for building responsive user interfaces, and we discuss some of these technologies (Section 9.5). We then provide an overview of web application architectures (Section 9.6) and cover performance issues in building large web applications (Section 9.7). Finally, we discuss issues in application security that are key to making applications resilient to attacks (Section 9.8), and encryption and its use in applications (Section 9.9).

9.1 Application Programs and User Interfaces

Although many people interact with databases, very few people use a query language to interact with a database system directly. The most common way in which users interact with databases is through an **application program** that provides a user interface at the front end and interfaces with a database at the back end. Such applications take input from users, typically through a forms-based interface, and either enter data into a database or extract information from a database based on the user input, and they then generate output, which is displayed to the user.

As an example of an application, consider a university registration system. Like other such applications, the registration system first requires you to identify and authenticate yourself, typically by a user name and password. The application then uses your

Telegram: @uni_k

www.konkur.in

identity to extract information, such as your name and the courses for which you have registered, from the database and displays the information. The application provides a number of interfaces that let you register for courses and query other information, such as course and instructor information. Organizations use such applications to automate a variety of tasks, such as sales, purchases, accounting and payroll, human-resources management, and inventory management, among many others.

Application programs may be used even when it is not apparent that they are being used. For example, a news site may provide a page that is transparently customized to individual users, even if the user does not explicitly fill any forms when interacting with the site. To do so, it actually runs an application program that generates a customized page for each user; customization can, for example, be based on the history of articles browsed by the user.

A typical application program includes a front-end component, which deals with the user interface, a backend component, which communicates with a database, and a middle layer, which contains "business logic," that is, code that executes specific requests for information or updates, enforcing rules of business such as what actions should be carried out to execute a given task or who can carry out what task.

Applications such as airline reservations have been around since the 1960s. In the early days of computer applications, applications ran on large "mainframe" computers, and users interacted with the application through terminals, some of which even supported forms. The growth of personal computers resulted in the development of database applications with graphical user interfaces, or GUIs. These interfaces depended on code running on a personal computer that directly communicated with a shared database. Such an architecture was called a *client–server architecture*. There were two drawbacks to using such applications: first, user machines had direct access to databases, leading to security risks. Second, any change to the application or the database required all the copies of the application, located on individual computers, to be updated together.

Two approaches have evolved to avoid the above problems:

- Web browsers provide a *universal front end*, used by all kinds of information services. Browsers use a standardized syntax, the **HyperText Markup Language (HTML)** standard, which supports both formatted display of information and creation of forms-based interfaces. The HTML standard is independent of the operating system or browser, and pretty much every computer today has a web browser installed. Thus a web-based application can be accessed from any computer that is connected to the internet.

  Unlike client–server architectures, there is no need to install any application-specific software on client machines in order to use web-based applications.

  However, sophisticated user interfaces, supporting features well beyond what is possible using plain HTML, are now widely used, and are built with the scripting language JavaScript, which is supported by most web browsers. JavaScript programs, unlike programs written in C, can be run in a safe mode, guaranteeing

Telegram: @uni_k

www.konkur.in

they cannot cause security problems. JavaScript programs are downloaded transparently to the browser and do not need any explicit software installation on the user's computer.

While the web browser provides the front end for user interaction, application programs constitute the back end. Typically, requests from a browser are sent to a web server, which in turn executes an application program to process the request. A variety of technologies are available for creating application programs that run at the back end, including Java servlets, Java Server Pages (JSP), Active Server Page (ASP), or scripting languages such as PHP and Python.

- Application programs are installed on individual devices, which are primarily mobile devices. They communicate with backend applications through an API and do not have direct access to the database. The back end application provides services, including user authentication, and ensures that users can only access services that they are authorized to access.

  This approach is widely used in mobile applications. One of the motivations for building such applications was to customize the display for the small screen of mobile devices. A second was to allow application code, which can be relatively large, to be downloaded or updated when the device is connected to a high-speed network, instead of downloading such code when a web page is accessed, perhaps over a lower bandwidth or more expensive mobile network.

With the increasing use of JavaScript code as part of web front ends, the difference between the two approaches above has today significantly decreased. The back end often provides an API that can be invoked from either mobile app or JavaScript code to carry out any required task at the back end. In fact, the same back end is often used to build multiple front ends, which could include web front ends with JavaScript, and multiple mobile platforms (primarily Android and iOS, today).

9.2 Web Fundamentals

In this section, we review some of the fundamental technology behind the World Wide Web, for readers who are not familiar with the technology underlying the web.

9.2.1 Uniform Resource Locators

A **uniform resource locator** (**URL**) is a globally unique name for each document that can be accessed on the web. An example of a URL is:

http://www.acm.org/sigmod

The first part of the URL indicates how the document is to be accessed: "http" indicates that the document is to be accessed by the **HyperText Transfer Protocol** ( ),

Telegram: @uni_k

www.konkur.in


```
<html>
<body>
<table border>
<tr> <th>ID</th>        <th>Name</th>     <th>Department</th> </tr>
<tr> <td>00128</td> <td>Zhang</td>    <td>Comp. Sci.</td> </tr>
<tr> <td>12345</td> <td>Shankar</td> <td>Comp. Sci.</td> </tr>
<tr> <td>19991</td> <td>Brandt</td>    <td>History</td> </tr>
</table>
</body>
</html>
```

Figure 9.1 Tabular data in HTML format.

which is a protocol for transferring HTML documents; "https" would indicate that the secure version of the HTTP protocol must be used, and is the preferred mode today. The second part gives the name of a machine that has a web server. The rest of the URL is the path name of the file on the machine, or other unique identifier of a document within the machine.

A URL can contain the identifier of a program located on the web server machine, as well as arguments to be given to the program. An example of such a URL is

https://www.google.com/search?q=silberschatz

which says that the program search on the server www.google.com should be executed with the argument q=silberschatz. On receiving a request for such a URL, the web server executes the program, using the given arguments. The program returns an HTML document to the web server, which sends it back to the front end.

9.2.2 HyperText Markup Language

Figure 9.1 is an example of a table represented in the HTML format, while Figure 9.2 shows the displayed image generated by a browser from the HTML representation of the table. The HTML source shows a few of the HTML tags. Every HTML page should be enclosed in an html tag, while the body of the page is enclosed in a body tag. A table

ID	Name	Department
00128	Zhang	Comp. Sci.
12345	Shankar	Comp. Sci.
19991	Brandt	History

Figure 9.2 Display of HTML source from Figure 9.1.

Telegram: @uni_k

www.konkur.in


```
<html>
<body>
<form action="PersonQuery" method=get>
Search for:
<select name="persontype">
      <option value="student" selected>Student </option>
      <option value="instructor"> Instructor </option>
</select> <br>
Name: <input type=text size=20 name="name">
<input type=submit value="submit">
</form>
</body>
</html>
```

Figure 9.3 An HTML form.

is specified by a table tag, which contains rows specified by a tr tag. The header row of the table has table cells specified by a th tag, while regular rows have table cells specified by a td tag. We do not go into more details about the tags here; see the bibliographical notes for references containing more detailed descriptions of HTML.

Figure 9.3 shows how to specify an HTML form that allows users to select the person type (student or instructor) from a menu and to input a number in a text box. Figure 9.4 shows how the above form is displayed in a web browser. Two methods of accepting input are illustrated in the form, but HTML also supports several other input methods. The action attribute of the form tag specifies that when the form is submitted (by clicking on the submit button), the form data should be sent to the URL PersonQuery (the URL is relative to that of the page). The web server is configured such that when this URL is accessed, a corresponding application program is invoked, with the user-provided values for the arguments persontype and name (specified in the select and input fields). The application program generates an HTML document, which is then sent back and displayed to the user; we shall see how to construct such programs later in this chapter.

HTTP defines two ways in which values entered by a user at the browser can be sent to the web server. The get method encodes the values as part of the URL. For example, if the Google search page used a form with an input parameter named q with the get

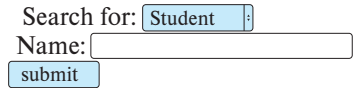


Figure 9.4 Display of HTML source from Figure 9.3.

Telegram: @uni_k

www.konkur.in

method, and the user typed in the string "silberschatz" and submitted the form, the browser would request the following URL from the web server:

https://www.google.com/search?q=silberschatz

The post method would instead send a request for the URL https://www.google.com, and send the parameter values as part of the HTTP protocol exchange between the web server and the browser. The form in Figure 9.3 specifies that the form uses the get method.

Although HTML code can be created using a plain text editor, there are a number of editors that permit direct creation of HTML text by using a graphical interface. Such editors allow constructs such as forms, menus, and tables to be inserted into the HTML document from a menu of choices, instead of manually typing in the code to generate the constructs.

HTML supports *stylesheets*, which can alter the default definitions of how an HTML formatting construct is displayed, as well as other display attributes such as background color of the page. The *cascading stylesheet* (CSS) standard allows the same stylesheet to be used for multiple HTML documents, giving a distinctive but uniform look to all the pages on a web site. You can find more information on stylesheets online, for example at www.w3schools.com/css/.

The HTML5 standard, which was released in 2014, provides a wide variety of form input types, including the following:

- Date and time selection, using <input type="date" name="abc">, and <input type="time" name="xyz">. Browsers would typically display a graphical date or time picker for such an input field; the input value is saved in the form attributes abc and xyz. The optional attributes min and max can be used to specify minimum and maximum values that can be chosen.
- File selection, using <input type="file", name="xyz">, which allows a file to be chosen, and its name saved in the form attribute xyz.
- Input restrictions (constraints) on a variety of input types, including minimum, maximum, format matching a regular expression, and so on. For example, <input type="number" name="start" min="0" max="55" step="5" value="0"> allows the user to choose one of 0, 5, 10, 15, and so on till 55, with a default value of 0.

9.2.3 Web Servers and Sessions

A **web server** is a program running on the server machine that accepts requests from a web browser and sends back results in the form of HTML documents. The browser and web server communicate via HTTP. Web servers provide powerful features, beyond the simple transfer of documents. The most important feature is the ability to execute

Telegram: @uni_k

www.konkur.in

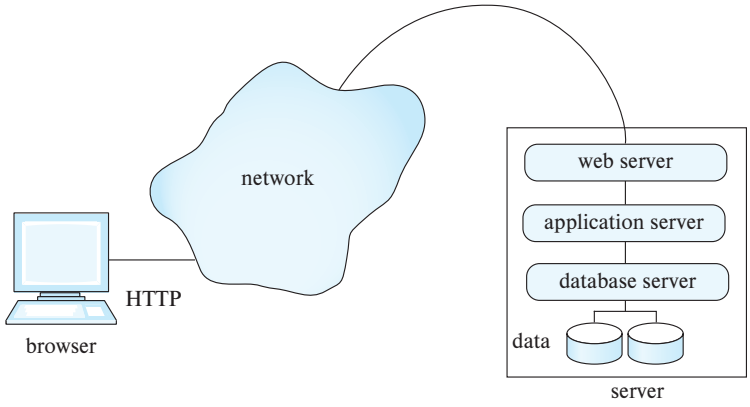


Figure 9.5 Three-layer web application architecture.

programs, with arguments supplied by the user, and to deliver the results back as an HTML document.

As a result, a web server can act as an intermediary to provide access to a variety of information services. A new service can be created by creating and installing an application program that provides the service. The **common gateway interface (CGI)** standard defines how the web server communicates with application programs. The application program typically communicates with a database server, through ODBC, JDBC, or other protocols, in order to get or store data.

Figure 9.5 shows a web application built using a three-layer architecture, with a web server, an application server, and a database server. Using multiple levels of servers increases system overhead; the CGI interface starts a new process to service each request, which results in even greater overhead.

Most web applications today therefore use a two-layer web application architecture, where the web and application servers are combined into a single server, as shown in Figure 9.6. We study systems based on the two-layer architecture in more detail in subsequent sections.

There is no continuous connection between the client and the web server; when a web server receives a request, a connection is temporarily created to send the request and receive the response from the web server. But the connection may then be closed, and the next request could come over a new connection. In contrast, when a user logs on to a computer, or connects to a database using ODBC or JDBC, a session is created, and session information is retained at the server and the client until the session is terminated—information such as the user-identifier of the user and session options that the user has set. One important reason that HTTP is **connectionless** is that most computers have limits on the number of simultaneous connections they can accommodate, and if a large number of sites on the web open connections to a single server, this limit would be exceeded, denying service to further users. With a connectionless protocol, the con-

Telegram: @uni_k

www.konkur.in

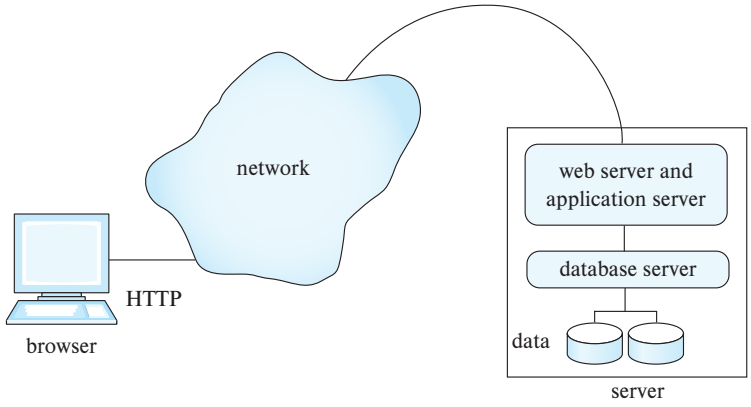


Figure 9.6 Two-layer web application architecture.

nection can be broken as soon as a request is satisfied, leaving connections available for other requests.[1]

Most web applications, however, need session information to allow meaningful user interaction. For instance, applications typically restrict access to information, and therefore need to authenticate users. Authentication should be done once per session, and further interactions in the session should not require reauthentication.

To implement sessions in spite of connections getting closed, extra information has to be stored at the client and returned with each request in a session; the server uses this information to identify that a request is part of a user session. Extra information about the session also has to be maintained at the server.

This extra information is usually maintained in the form of a **cookie** at the client; a cookie is simply a small piece of text containing identifying information and with an associated name. For example, google.com may set a cookie with the name prefs, which encodes preferences set by the user such as the preferred language and the number of answers displayed per page. On each search request, google.com can retrieve the cookie named prefs from the user's browser, and display results according to the specified preferences. A domain (web site) is permitted to retrieve only cookies that it has set, not cookies set by other domains, and cookie names can be reused across domains.

For the purpose of tracking a user session, an application may generate a session identifier (usually a random number not currently in use as a session identifier), and send a cookie named (for instance) sessionid containing the session identifier. The session identifier is also stored locally at the server. When a request comes in, the application server requests the cookie named sessionid from the client. If the client

[1]For performance reasons, connections may be kept open for a short while, to allow subsequent requests to reuse the connection. However, there is no guarantee that the connection will be kept open, and applications must be designed assuming the connection may be closed as soon as a request is serviced.

Telegram: @uni_k

www.konkur.in

does not have the cookie stored, or returns a value that is not currently recorded as a valid session identifier at the server, the application concludes that the request is not part of a current session. If the cookie value matches a stored session identifier, the request is identified as part of an ongoing session.

If an application needs to identify users securely, it can set the cookie only after authenticating the user; for example a user may be authenticated only when a valid user name and password are submitted.[2]

For applications that do not require high security, such as publicly available news sites, cookies can be stored permanently at the browser and at the server; they identify the user on subsequent visits to the same site, without any identification information being typed in. For applications that require higher security, the server may invalidate (drop) the session after a time-out period, or when the user logs out. (Typically a user logs out by clicking on a logout button, which submits a logout form, whose action is to invalidate the current session.) Invalidating a session merely consists of dropping the session identifier from the list of active sessions at the application server.

9.3 Servlets

The Java **servlet** specification defines an application programming interface for communication between the web/application server and the application program. The `HttpServlet` class in Java implements the servlet API specification; servlet classes used to implement specific functions are defined as subclasses of this class.[3] Often the word *servlet* is used to refer to a Java program (and class) that implements the servlet interface. Figure 9.7 shows a servlet example; we explain it in detail shortly.

The code for a servlet is loaded into the web/application server when the server is started, or when the server receives a remote HTTP request to execute a particular servlet. The task of a servlet is to process such a request, which may involve accessing a database to retrieve necessary information, and dynamically generating an HTML page to be returned to the client browser.

9.3.1 A Servlet Example

Servlets are commonly used to generate dynamic responses to HTTP requests. They can access inputs provided through HTML forms, apply "business logic" to decide what

[2]The user identifier could be stored at the client end, in a cookie named, for example, `userid`. Such cookies can be used for low-security applications, such as free web sites identifying their users. However, for applications that require a higher level of security, such a mechanism creates a security risk: The value of a cookie can be changed at the browser by a malicious user, who can then masquerade as a different user. Setting a cookie (named `sessionid`, for example) to a randomly generated session identifier (from a large space of numbers) makes it highly improbable that a user can masquerade as (i.e., pretend to be) another user. A sequentially generated session identifier, on the other hand, would be susceptible to masquerading.

[3]The servlet interface can also support non-HTTP requests,

Telegram: @uni_k

www.konkur.in

response to provide, and then generate HTML output to be sent back to the browser. Servlet code is executed on a web or application server.

Figure 9.7 shows an example of servlet code to implement the form in Figure 9.3. The servlet is called PersonQueryServlet, while the form specifies that "action="PersonQuery"." The web/application server must be told that this servlet is to be used to handle requests for PersonQuery, which is done by using the anno-

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

@WebServlet("PersonQuery")
public class PersonQueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        ... check if user is logged in ...
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");

        String persontype = request.getParameter("persontype");
        String name = request.getParameter("name");
        if(persontype.equals("student")) {
            ... code to find students with the specified name ...
            ... using JDBC to communicate with the database ..
            ... Assume ResultSet rs has been retrieved, and
            ... contains attributes ID, name, and department name
            String headers = new String[]{"ID", "Name", "Department Name"};
            Util::resultSetToHTML(rs, headers, out);
        }
        else {
            ... as above, but for instructors ...
        }
        out.println("</BODY>");
        out.close();
    }
}
```

Figure 9.7 Example of servlet code.

Telegram: @uni_k

www.konkur.in

tation @WebServlet("PersonQuery") shown in the code. The form specifies that the HTTP get mechanism is used for transmitting parameters. So the doGet() method of the servlet, as defined in the code, is invoked.

Each servlet request results in a new thread within which the call is executed, so multiple requests can be handled in parallel. Any values from the form menus and input fields on the web page, as well as cookies, pass through an object of the HttpServletRequest class that is created for the request, and the reply to the request passes through an object of the class HttpServletResponse.

The doGet() method in the example extracts values of the parameters persontype and name by using request.getParameter(), and uses these values to run a query against a database. The code used to access the database and to get attribute values from the query result is not shown; refer to Section 5.1.1.5 for details of how to use JDBC to access a database. We assume that the result of the query in the form of a JDBC ResultSet is available in the variable resultset.

The servlet code returns the results of the query to the requester by outputting them to the HttpServletResponse object response. Outputting the results to response is implemented by first getting a PrintWriter object out from response, and then printing the query result in HTML format to out. In our example, the query result is printed by calling the function Util::resultSetToHTML(resultset, header, out), which is shown in Figure 9.8. The function uses JDBC metadata function on the ResultSet rs to figure out how many columns need to be printed. An array of column headers is passed to this function to be printed out; the column names could have been obtained using JDBC metadata, but the database column names may not be appropriate for display to a user, so we provide meaningful column names to the function.

9.3.2 Servlet Sessions

Recall that the interaction between a browser and a web/application server is stateless. That is, each time the browser makes a request to the server, the browser needs to connect to the server, request some information, then disconnect from the server. Cookies can be used to recognize that a request is from the same browser session as an earlier request. However, cookies form a low-level mechanism, and programmers require a better abstraction to deal with sessions.

The servlet API provides a method of tracking a session and storing information pertaining to it. Invocation of the method getSession(false) of the class HttpServletRequest retrieves the HttpSession object corresponding to the browser that sent the request. An argument value of true would have specified that a new session object must be created if the request is a new request.

When the getSession() method is invoked, the server first asks the client to return a cookie with a specified name. If the client does not have a cookie of that name, or returns a value that does not match any ongoing session, then the request is not part of an ongoing session. In this case, getSession() would return a null value, and the servlet could direct the user to a login page.

Telegram: @uni_k

www.konkur.in


```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Util {
    public static void resultSetToHTML(ResultSet rs,
                                       String headers[], PrintWriter out) {
        ResultSetMetaData rsmd = rs.getMetaData();
        int numCols = rsmd.getColumnCount();
        out.println("<table border=1>");
        out.println("<tr>");
        for (int i=0; i < numCols; i++)
            out.println("<th>"+ headers[i] + </th>
        out.println("</tr>");
        while (rs.next()) {
            out.println("<tr>");
            for (int i=0; i < numCols; i++)
                out.println("<td>"+ rs.getString(i) + </td>
            out.println("</tr>");
        }
    }
}
```

Figure 9.8 Utility function to output ResultSet as a table.

The login page could allow the user to provide a user name and password. The servlet corresponding to the login page could verify that the password matches the user; for example, by using the user name to retrieve the password from the database and checking if the password entered matches the stored password.[4]

If the user is properly authenticated, the login servlet would execute getSession(true), which would return a new session object. To create a new session, the server would internally carry out the following tasks: set a cookie (called, for example, sessionId) with a session identifier as its associated value at the client browser, create a new session object, and associate the session identifier value with the session object.

[4]It is a bad idea to store unencrypted passwords in the database, since anyone with access to the database contents, such as a system administrator or a hacker, could steal the password. Instead, a hashing function is applied to the password, and the result is stored in the database; even if someone sees the hashing result stored in the database, it is very hard to infer what was the original password. The same hashing function is applied to the user-supplied password, and the result is compared with the stored hashing result. Further, to ensure that even if two users use the same password the hash values are different, the password system typically stores a different random string (called the *salt*) for each user, and it appends the random string to the password before computing the hash value. Thus, the password relation would have the schema *user_password*(*user, salt, passwordhash*), where *passwordhash* is generated by *hash*(*append*(*password*,*salt*)). Encryption is described in more detail in Section 9.9.1.

Telegram: @uni_k

www.konkur.in

The servlet code can also store and look up (attribute-name, value) pairs in the HttpSession object, to maintain state across multiple requests within a session. For example, after the user is authenticated and the session object has been created, the login servlet could store the user-id of the user as a session parameter by executing the method

```
session.setAttribute("userid", userid)
```

on the session object returned by getSession(); the Java variable userid is assumed to contain the user identifier.

If the request was part of an ongoing session, the browser would have returned the cookie value, and the corresponding session object would be returned by getSession(). The servlet could then retrieve session parameters such as user-id from the session object by executing the method

```
session.getAttribute("userid")
```

on the session object returned above. If the attribute userid is not set, the function would return a null value, which would indicate that the client user has not been authenticated.

Consider the line in the servlet code in Figure 9.7 that says "... check if user is logged in...". The following code implements this check; if the user is not logged in, it sends an error message, and after a gap of 5 seconds, redirects the user to the login page.

```
Session session = request.getSession(false);
if (session == null || session.getAttribute(userid) == null) {
      out.println("You are not logged in.");
      response.setHeader("Refresh", "5;url=login.html");
      return();
}
```

9.3.3 Servlet Life Cycle

The life cycle of a servlet is controlled by the web/application server in which the servlet has been deployed. When there is a client request for a specific servlet, the server first checks if an instance of the servlet exists or not. If not, the server loads the servlet class into the Java virtual machine (JVM) and creates an instance of the servlet class. In addition, the server calls the init() method to initialize the servlet instance. Notice that each servlet instance is initialized only once when it is loaded.

After making sure the servlet instance does exist, the server invokes the service method of the servlet, with a request object and a response object as parameters. By default, the server creates a new thread to execute the method; thus, multiple

Telegram: @uni_k

www.konkur.in

requests on a servlet can execute in parallel, without having to wait for earlier requests to complete execution. The service method calls doGet or doPost as appropriate.

When no longer required, a servlet can be shut down by calling the destroy() method. The server can be set up to shut down a servlet automatically if no requests have been made on a servlet within a time-out period; the time-out period is a server parameter that can be set as appropriate for the application.

9.3.4 Application Servers

Many application servers provide built-in support for servlets. One of the most popular is the Tomcat Server from the Apache Jakarta Project. Other application servers that support servlets include Glassfish, JBoss, BEA Weblogic Application Server, Oracle Application Server, and IBM's WebSphere Application Server.

The best way to develop servlet applications is by using an IDE such as Eclipse or NetBeans, which come with Tomcat or Glassfish servers built in.

Application servers usually provide a variety of useful services, in addition to basic servlet support. They allow applications to be deployed or stopped, and they provide functionality to monitor the status of the application server, including performance statistics. Many application servers also support the Java 2 Enterprise Edition (J2EE) platform, which provides support and APIs for a variety of tasks, such as for handling objects, and parallel processing across multiple application servers.

9.4 Alternative Server-Side Frameworks

There are several alternatives to Java Servlets for processing requests at the application server, including scripting languages and web application frameworks developed for languages such as Python.

9.4.1 Server-Side Scripting

Writing even a simple web application in a programming language such as Java or C is a time-consuming task that requires many lines of code and programmers who are familiar with the intricacies of the language. An alternative approach, that of **server-side scripting**, provides a much easier method for creating many applications. Scripting languages provide constructs that can be embedded within HTML documents.

In server-side scripting, before delivering a web page, the server executes the scripts embedded within the HTML contents of the page. Each piece of script, when executed, can generate text that is added to the page (or may even delete content from the page). The source code of the scripts is removed from the page, so the client may not even be aware that the page originally had any code in it. The executed script may also contain SQL code that is executed against a database. Many of these languages come with libraries and tools that together constitute a framework for web application development.

Telegram: @uni_k

www.konkur.in


```
<html>
<head> <title> Hello </title> </head>
<body>
 < % if (request.getParameter("name") == null)
   { out.println("Hello World"); }
   else { out.println("Hello, " + request.getParameter("name")); }
 %>
</body>
</html>
```

Figure 9.9 A JSP page with embedded Java code.

Some of the widely used scripting frameworks include Java Server Pages (JSP), ASP.NET from Microsoft, PHP, and Ruby on Rails. These frameworks allow code written in languages such as Java, C#, VBScript, and Ruby to be embedded into or invoked from HTML pages. For instance, JSP allows Java code to be embedded in HTML pages, while Microsoft's ASP.NET and ASP support embedded C# and VBScript.

9.4.1.1 Java Server Pages

Next we briefly describe **Java Server Pages** (**JSP**), a scripting language that allows HTML programmers to mix static HTML with dynamically generated HTML. The motivation is that, for many dynamic web pages, most of their content is still static (i.e., the same content is present whenever the page is generated). The dynamic content of the web pages (which are generated, for example, on the basis of form parameters) is often a small part of the page. Creating such pages by writing servlet code results in a large amount of HTML being coded as Java strings. JSP instead allows Java code to be embedded in static HTML; the embedded Java code generates the dynamic part of the page. JSP scripts are actually translated into servlet code that is then compiled, but the application programmer is saved the trouble of writing much of the Java code to create the servlet.

Figure 9.9 shows the source text of a JSP page that includes embedded Java code. The Java code in the script is distinguished from the surrounding HTML code by being enclosed in <% … %>. The code uses `request.getParameter()` to get the value of the attribute `name`.

When a JSP page is requested by a browser, the application server generates HTML output from the page, which is sent to the browser. The HTML part of the JSP page is output as is.[5] Wherever Java code is embedded within <% …%>, the code is replaced in the HTML output by the text it prints to the object `out`. In the JSP code in Figure 9.9,

[5] JSP allows a more complex embedding, where HTML code is within a Java if-else statement, and gets output conditionally depending on whether the if condition evaluates to true or not.

Telegram: @uni_k

www.konkur.in

if no value was entered for the form parameter name, the script prints "Hello World"; if a value was entered, the script prints "Hello" followed by the name.

A more realistic example may perform more complex actions, such as looking up values from a database using JDBC.

JSP also supports the concept of a *tag library*, which allows the use of tags that look much like HTML tags but are interpreted at the server and are replaced by appropriately generated HTML code. JSP provides a standard set of tags that define variables and control flow (iterators, if-then-else), along with an expression language based on JavaScript (but interpreted at the server). The set of tags is extensible, and a number of tag libraries have been implemented. For example, there is a tag library that supports paginated display of large data sets and a library that simplifies display and parsing of dates and times.

9.4.1.2 PHP

PHP is a scripting language that is widely used for server-side scripting. PHP code can be intermixed with HTML in a manner similar to JSP. The characters "<?php" indicate the start of PHP code, while the characters "?>" indicate the end of PHP code. The following code performs the same actions as the JSP code in Figure 9.9.

```
<html>
<head> <title> Hello </title> </head>
<body>
  <?php if (!isset($_REQUEST['name']))
    { echo 'Hello World'; }
    else { echo 'Hello, ' . $_REQUEST['name']; }
  ?>
</body>
</html>
```

The array $_REQUEST contains the request parameters. Note that the array is indexed by the parameter name; in PHP arrays can be indexed by arbitrary strings, not just numbers. The function isset checks if the element of the array has been initialized. The echo function prints its argument to the output HTML. The operator "." between two strings concatenates the strings.

A suitably configured web server would interpret any file whose name ends in ".php" to be a PHP file. If the file is requested, the web server processes it in a manner similar to how JSP files are processed and returns the generated HTML to the browser.

A number of libraries are available for the PHP language, including libraries for database access using ODBC (similar to JDBC in Java).

9.4.2 Web Application Frameworks

Web application development frameworks ease the task of constructing web applications by providing features such as these:

Telegram: @uni_k

www.konkur.in

- A library of functions to support HTML and HTTP features, including sessions.
- A template scripting system.
- A controller that maps user interaction events such as form submits to appropriate functions that handle the event. The controller also manages authentication and sessions. Some frameworks also provide tools for managing authorizations.
- A (relatively) declarative way of specifying a form with validation constraints on user inputs, from which the system generates HTML and Javascript/Ajax code to implement the form.
- An object-oriented model with an object-relational mapping to store data in a relational database (described in Section 9.6.2).

Thus, these frameworks provide a variety of features that are required to build web applications in an integrated manner. By generating forms from declarative specifications and managing data access transparently, the frameworks minimize the amount of coding that a web application programmer has to carry out.

There are a large number of such frameworks, based on different languages. Some of the more widely used frameworks include the Django framework for the Python language, Ruby on Rails, which supports the Rails framework on the Ruby programming language, Apache Struts, Swing, Tapestry, and WebObjects, all based on Java/JSP. Many of these frameworks also make it easy to create simple **CRUD** web interfaces; that is, interfaces that support create, read, update and delete of objects/tuples by generating code from an object model or a database. Such tools are particularly useful to get simple applications running quickly, and the generated code can be edited to build more sophisticated web interfaces.

9.4.3 The Django Framework

The Django framework for Python is a widely used web application framework. We illustrate a few features of the framework through examples.

Views in Django are functions that are equivalent to servlets in Java. Django requires a mapping, typically specified in a file `urls.py`, which maps URLs to Django views. When the Django application server receives an HTTP request, it uses the URL mapping to decide which view function to invoke.

Figure 9.10 shows sample code implementing the person query task that we earlier implemented using Java servlets. The code shows a view called `person_query_view`. We assume that the `PersonQuery` URL is mapped to the view `person_query_view`, and is invoked from the HTML form shown earlier in Figure 9.3.

We also assume that the root of the application is mapped to a `login_view`. We have not shown the code for `login_view`, but we assume it displays a login form, and on submit it invokes the `authenticate` view. We have not shown the `authenticate` view,

Telegram: @uni_k

www.konkur.in

either, but we assume that it checks the login name and password. If the password is validated, the authenticate view redirects to a person_query_form, which displays the HTML code that we saw earlier in Figure 9.3; if password validation fails, it redirects to the login_view.

Returning to Figure 9.10, the view person_query_view() first checks if the user is logged in by checking the session variable username. If the session variable is not set, the browser is redirected to the login screen. Otherwise, the requested user informa-

```
from django.http import HttpResponse
from django.db import connection

def result_set_to_html(headers, cursor):
    html = "<table border=1>"
    html += "<tr>"
    for header in headers:
        html += "<th>" + header + "</th>"
    html += "</tr>"
    for row in cursor.fetchall():
        html += "<tr>"
        for col in row:
            html += "<td>" + col + "</td>"
        html += "</tr>"
    html += "</table>"
    return html

def person_query_view(request):
    if "username" not in request.session:
        return login_view(request)
    persontype = request.GET.get("persontype")
    personname = request.GET.get("personname")
    if persontype == "student":
        query_tmpl = "select id, name, dept_name from student where name=%s"
    else:
        query_tmpl = "select id, name, dept_name from instructor where name=%s"
    with connection.cursor() as cursor:
        cursor.execute(query_tmpl, [personname])
        headers = ["ID", "Name", "Department Name"]
        return HttpResponse(result_set_to_html(headers, cursor))
```

Figure 9.10 The person query application in Django.

Telegram: @uni_k

www.konkur.in

tion is fetched by connecting to the database; connection details for the database are specified in a Django configuration file settings.py and are omitted in our description. A cursor (similar to a JDBC statement) is opened on the connection, and the query is executed using the cursor. Note that the first argument of cursor.execute is the query, with parameters marked by "%s", and the second argument is a list of values for the parameters. The result of the database query is then displayed by calling a function result_set_to_html(), which iterates over the result set fetched from the database and outputs the results in HTML format to a string; the string is then returned as an HttpResponse.

Django provides support for a number of other features, such as creating HTML forms and validating data entered in the forms, annotations to simplify checking of authentication, and templates for creating HTML pages, which are somewhat similar to JSP pages. Django also supports an object-relation mapping system, which we describe in Section 9.6.2.2.

9.5 Client-Side Code and Web Services

The two most widely used classes of user interfaces today are the web interfaces and mobile application interfaces.

While early generation web browsers only displayed HTML code, the need was soon felt to allow code to run on the browsers. **Client-side scripting languages** are languages designed to be executed on the client's web browser. The primary motivation for such scripting languages is flexible interaction with the user, providing features beyond the limited interaction power provided by HTML and HTML forms. Further, executing programs at the client site speeds up interaction greatly compared to every interaction being sent to a server site for processing.

The **JavaScript** language is by far the most widely used client-side scripting language. The current generation of web interfaces uses the JavaScript scripting language extensively to construct sophisticated user interfaces.

Any client-side interface needs to store and retrieve data from the back end. Directly accessing a database is not a good idea, since it not only exposes low-level details, but it also exposes the database to attacks. Instead, back ends provide access to store and retrieve data through web services. We discuss web services in Section 9.5.2.

Mobile applications are very widely used, and user interfaces for mobile devices are very important today. Although we do not cover mobile application development in this book, we offer pointers to some mobile application development frameworks in Section 9.5.4.

9.5.1 JavaScript

JavaScript is used for a variety of tasks, including validation, flexible user interfaces, and interaction with web services, which we now describe.

Telegram: @uni_k

www.konkur.in

9.5.1.1 Input Validation

Functions written in JavaScript can be used to perform error checks (validation) on user input, such as a date string being properly formatted, or a value entered (such as age) being in an appropriate range. These checks are carried out on the browser as data are entered even before the data are sent to the web server.

With HTML5, many validation constraints can be specified as part of the input tag. For example, the following HTML code:

```
<input type="number" name="credits" size="2" min="1" max="15">
```

ensures that the input for the parameter "credits" is a number between 1 and 15. More complex validations that cannot be performed using HTML5 features are best done using JavaScript.

Figure 9.11 shows an example of a form with a JavaScript function used to validate a form input. The function is declared in the head section of the HTML document. The form accepts a start and an end date. The validation function ensures that the start date

```
<html>
<head>
<script type="text/javascript">
      function validate() {
            var startdate = new Date (document.getElementById("start").value);
            var enddate = new Date (document.getElementById("end").value);
            if(startdate > enddate) {
                  alert("Start date is > end date");
                  return false;
            }
  }
</script>
</head>

<body>
<form action="submitDates" onsubmit="return validate()">
      Start Date: <input type="date" id="start"><br />
      End Date : <input type="date" id="end"><br />
      <input type="submit" value="Submit">
</form>
</body>
</html>
```

Figure 9.11 Example of JavaScript used to validate form input.

Telegram: @uni_k

www.konkur.in

is not greater than the end date. The form tag specifies that the validation function is to be invoked when the form is submitted. If the validation fails, an alert box is shown to the user, and if it succeeds, the form is submitted to the server.

9.5.1.2 Responsive User Interfaces

The most important benefit of JavaScript is the ability to create highly responsive user interfaces within a browser using JavaScript. The key to building such a user interface is the ability to dynamically modify the HTML code being displayed by using JavaScript. The browser parses HTML code into an in-memory tree structure defined by a standard called the **Document Object Model** (**DOM**). JavaScript code can modify the tree structure to carry out certain operations. For example, suppose a user needs to enter a number of rows of data, for example multiple items in a single bill. A table containing text boxes and other form input methods can be used to gather user input. The table may have a default size, but if more rows are needed, the user may click on a button labeled (for example) "Add Item." This button can be set up to invoke a JavaScript function that modifies the DOM tree by adding an extra row in the table.

Although the JavaScript language has been standardized, there are differences between browsers, particularly in the details of the DOM model. As a result, JavaScript code that works on one browser may not work on another. To avoid such problems, it is best to use a JavaScript library, such as the JQuery library, which allows code to be written in a browser-independent way. Internally, the functions in the library can find out which browser is in use and send appropriately generated JavaScript to the browser.

JavaScript libraries such as JQuery provide a number of UI elements, such as menus, tabs, widgets such as sliders, and features such as autocomplete, that can be created and executed using library functions.

The HTML5 standard supports a number of features for rich user interaction, including drag-and-drop, geolocation (which allows the user's location to be provided to the application with user permission), allowing customization of the data/interface based on location. HTML5 also supports Server-Side Events (SSE), which allows a backend to notify the front end when some event occurs.

9.5.1.3 Interfacing with Web Services

Today, JavaScript is widely used to create dynamic web pages, using several technologies that are collectively called **Ajax**. Programs written in JavaScript can communicate with the web server asynchronously (that is, in the background, without blocking user interaction with the web browser), and can fetch data and display it. The *JavaScript Object Notation*, or JSON, representation described in Section 8.1.2 is the most widely used data format for transferring data, although other formats such as XML are also used.

The role of the code for the above tasks, which runs at the application server, is to send data to the JavaScript code, which then renders the data on the browser.

Telegram: @uni_k

www.konkur.in

Such backend services, which serve the role of functions which can be invoked to fetch required data, are known as *web services*. Such services can be implemented using Java Servlets, Python, or any of a number of other language frameworks.

As an example of the use of Ajax, consider the autocomplete feature implemented by many web applications. As the user types a value in a text box, the system suggests completions for the value being typed. Such autocomplete is very useful for helping a user choose a value from a large number of values where a drop-down list would not be feasible. Libraries such as jQuery provide support for autocomplete by associating a function with a text box; the function takes partial input in the box, connected to a web back end to get possible completions, and displays them as suggestions for the autocomplete.

The JavaScript code shown in Figure 9.12 uses the jQuery library to implement autocomplete and the DataTables plug-in for the jQuery library to provide a tabular display of data. The HTML code has a text input box for name, which has an id attribute set to name. The script associates an autocomplete function from the jQuery library with the text box by using $("#name") syntax of jQuery to locate the DOM node for text box with id "name", and then associating the autocomplete function with the node. The attribute source passed to the function identifies the web service that must be invoked to get values for the autocomplete functionality. We assume that a servlet /autocomplete_name has been defined, which accepts a parameter term containing the letters typed so far by the user, even as they are being typed. The servlet should return a JSON array of names of students/instructors that match the letters in the term parameter.

The JavaScript code also illustrates how data can be retrieved from a web service and then displayed. Our sample code uses the DataTables jQuery plug-in; there are a number of other alternative libraries for displaying tabular data. We assume that the person_query_ajax Servlet, which is not shown, returns the ID, name, and department name of students or instructors with a given name, as we saw earlier in Figure 9.7, but encoded in JSON as an object with attribute data containing an array of rows; each row is a JSON object with attributes id, name, and dept_name.

The line starting with myTable shows how the jQuery plug-in DataTable is associated with the HTML table shown later in the figure, whose identifier is personTable. When the button "Show details" is clicked, the function loadTableAsync() is invoked. This function first creates a URL string url that is used to invoke person_query_ajax with values for person type and name. The function ajax.url(url).load() invoked on myTable fills the rows of the table using the JSON data fetched from the web service whose URL we created above. This happens asynchronously; that is, the function returns immediately, but when the data have been fetched, the table rows are filled with the returned data.

Figure 9.13 shows a screenshot of a browser displaying the result of the code in Figure 9.12.

As another example of the use of Ajax, consider a web site with a form that allows you to select a country, and once a country has been selected, you are allowed to select

Telegram: @uni_k

www.konkur.in


```
<html> <head>
<script src="https://code.jquery.com/jquery-3.3.1.js"> </script>
<script src="https://cdn.datatables.net/1.10.19/js/jquery.dataTables.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>
<script src="https://cdn.datatables.net/1.10.19/js/jquery.dataTables.min.js"></script>
<link rel="stylesheet"
    href="https://code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css" />
<link rel="stylesheet"
    href="https://cdn.datatables.net/1.10.19/css/jquery.dataTables.min.css"/>
<script>
    var myTable;
    $(document).ready(function() {
        $("#name").autocomplete({ source: "/autocomplete_name" });
        myTable = $("#personTable").DataTable({
        columns: [{data:"id"}, {data:"name"}, {data:"dept_name"}]
        });
    });
    function loadTableAsync() {
        var params = {persontype:$("#persontype").val(), name:$("#name").val()};
        var url = "/person_query_ajax?" + jQuery.param(params);
        myTable.ajax.url(url).load();
    }
</script>
</head> <body>
Search for:
<select id="persontype">
    <option value="student" selected>Student </option>
    <option value="instructor"> Instructor </option>
</select> <br>
Name: <input type=text size=20 id="name">
<button onclick="loadTableAsync()"> Show details </button>
<table id="personTable" border="1">
    <thead>
        <tr> <th>ID</th> <th>Name</th> <th>Dept. Name</th> </tr>
    </thead>
</table>
</body> </html>
```

Figure 9.12 HTML page using JavaScript and Ajax.

a state from a list of states in that country. Until the country is selected, the drop-down list of states is empty. The Ajax framework allows the list of states to be downloaded

Telegram: @uni_k

www.konkur.in

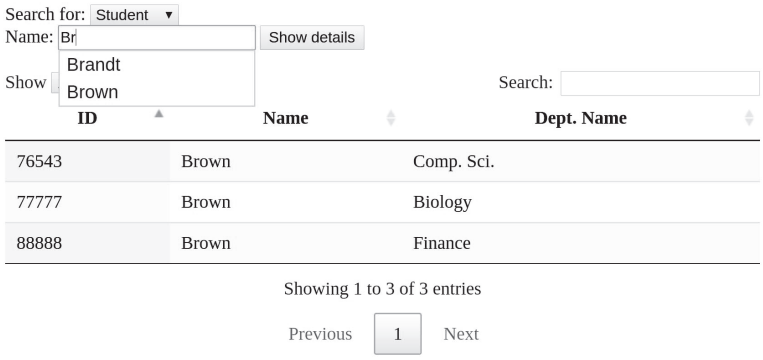


Figure 9.13 Screenshot of display generated by Figure 9.12.

from the web site in the background when the country is selected, and as soon as the list has been fetched, it is added to the drop-down list, which allows you to select the state.

9.5.2 Web Services

A **web service** is an application component that can be invoked over the web and functions, in effect, like an application programming interface. A web service request is sent using the HTTP protocol, it is executed at an application server, and the results are sent back to the calling program.

Two approaches are widely used to implement web services. In the simpler approach, called **Representation State Transfer** (or **REST**), web service function calls are executed by a standard HTTP request to a URL at an application server, with parameters sent as standard HTTP request parameters. The application server executes the request (which may involve updating the database at the server), generates and encodes the result, and returns the result as the result of the HTTP request. The most widely used encoding for the results today is the JSON representation, although XML, which we saw earlier in Section 8.1.3, is also used. The requestor parses the returned page to access the returned data.

In many applications of such RESTful web services (i.e., web services using REST), the requestor is JavaScript code running in a web browser; the code updates the browser screen using the result of the function call. For example, when you scroll the display on a map interface on the web, the part of the map that needs to be newly displayed may be fetched by JavaScript code using a RESTful interface and then displayed on the screen.

While some web services are not publicly documented and are used only internally by specific applications, other web services have their interfaces documented and can be used by any application. Such services may allow use without any restriction,

Telegram: @uni_k

www.konkur.in

may require users to be logged in before accessing the service, or may require users or application developers to pay the web service provider for the privilege of using the service.

Today, a very large variety of RESTful web services are available, and most front-end applications use one or more such services to perform backend activities. For example, your web-based email system, your social media web page, or your web-based map service would almost surely be built with JavaScript code for rendering and would use backend web services to fetch data as well as to perform updates. Similarly, any mobile app that stores data at the back end almost surely uses web services to fetch data and to perform updates.

Web services are also increasingly used at the backend, to make use of functionalities provided by other backend systems. For example, web-based storage systems provide a web service API for storing and retrieving data; such services are provided by a number of providers, such as Amazon S3, Google Cloud Storage, and Microsoft Azure. They are very popular with application developers since they allow storage of very large amounts of data, and they support a very large number of operations per second, allowing scalability far beyond what a centralized database can support.

There are many more such web-service APIs. For example, text-to-speech, speech recognition, and vision web-service APIs allow developers to construct applications incorporating speech and image recognition with very little development effort.

A more complex and less frequently used approach, sometimes referred to as "Big Web Services," uses XML encoding of parameters as well as results, has a formal definition of the web API using a special language, and uses a protocol layer built on top of the HTTP protocol.

9.5.3 Disconnected Operation

Many applications wish to support some operations even when a client is disconnected from the application server. For example, a student may wish to complete an application form even if her laptop is disconnected from the network but have it saved back when the laptop is reconnected. As another example, if an email client is built as a web application, a user may wish to compose an email even if her laptop is disconnected from the network and have it sent when it is reconnected. Building such applications requires local storage in the client machine.

The HTML5 standard supports local storage, which can be accessed using JavaScript. The code:

```
if (typeof(Storage) !== "undefined") { // browser supports local storage
...
}
```

checks if the browser supports local storage. If it does, the following functions can be used to store, load, or delete values for a given key.

Telegram: @uni_k

www.konkur.in


```
localStorage.setItem(key, value)
localStorage.getItem(key)
localStorage.deleteItem(key)
```

To avoid excessive data storage, the browser may limit a web site to storing at most some amount of data; the default maximum is typically 5 megabytes.

The above interface only allows storage/retrieval of key/value pairs. Retrieval requires that a key be provided; otherwise the entire set of key/value pairs will need to be scanned to find a required value. Applications may need to store tuples indexed on multiple attributes, allowing efficient access based on values of any of the attributes. HTML5 supports IndexedDB, which allows storage of JSON objects with indices on multiple attributes. IndexedDB also supports schema versions and allows the developer to provide code to migrate data from one schema version to the next version.

9.5.4 Mobile Application Platforms

Mobile applications (or mobile apps, for short) are widely used today, and they form the primary user interface for a large class of users. The two most widely used mobile platforms today are Android and iOS. Each of these platforms provides a way of building applications with a graphical user interface, tailored to small touch-screen devices. The graphical user interface provides a variety of standard GUI features such as menus, lists, buttons, check boxes, progress bars, and so on, and the ability to display text, images, and video.

Mobile apps can be downloaded and stored and used later. Thus, the user can download apps when connected to a high-speed network and then use the app with a lower-speed network. In contrast, web apps may get downloaded when they are used, resulting in a lot of data transfer when a user may be connected to a lower-speed network or a network where data transfer is expensive. Further, mobile apps can be better tuned to small-sized devices than web apps, with user interfaces that work well on small devices. Mobile apps can also be compiled to machine code, resulting in lower power demands than web apps. More importantly, unlike (earlier generation) web apps, mobile apps can store data locally, allowing offline usage. Further, mobile apps have a well-developed authorization model, allowing them to use information and device features such as location, cameras, contacts, and so on with user authorization.

However, one of the drawbacks of using mobile-app interfaces is that code written for the Android platform can only run on that platform and not on iOS, and vice versa. As a result, developers are forced to code every application twice, once for Android and once for iOS, unless they decide to ignore one of the platforms completely, which is not very desirable.

The ability to create applications where the same high-level code can run on either Android or iOS is clearly very important. The *React Native framework* based on JavaScript, developed by Facebook, and the *Flutter framework* based on the *Dart* language developed by Google, are designed to allow cross-platform development. (Dart

Telegram: @uni_k

www.konkur.in

is a language optimized for developing user interfaces, providing features such as asynchronous function invocation and functions on streams.) Both frameworks allow much of the application code to be common for both Android and iOS, but some functionality can be made specific to the underlying platform in case it is not supported in the platform-independent part of the framework.

With the wide availability of high-speed mobile networks, some of the motivation for using mobile apps instead of web apps, such as the ability to download ahead of time, is not as important anymore. A new generation of web apps, called **Progressive Web Apps** (**PWA**) that combine the benefits of mobile apps with web apps is seeing increasing usage. Such apps are built using JavaScript and HTML5 and are tailored for mobile devices.

A key enabling feature for PWAs is the HTML5 support for local data storage, which allows apps to be used even when the device is offline. Another enabling feature is the support for compilation of JavaScript code; compilation is restricted to code that follows a restricted syntax, since compilation of arbitrary JavaScript code is not practical. Such compilation is typically done just-in-time, that is, it is done when the code needs to be executed, or if it has already been executed multiple times. Thus, by writing CPU-heavy parts of a web application using only JavaScript features that allow compilation, it is possible to ensure CPU and energy-efficient execution of the code on a mobile device.

PWAs also make use of HTML5 service workers, which allow a script to run in the background in the browser, separate from a web page. Such service workers can be used to perform background synchronization operations between the local store and a web service, or to receive or push notifications from a backend service. HTML5 also allows apps to get device location (after user authorization), allowing PWAs to use location information.

Thus, PWAs are likely to see increasing use, replacing many (but certainly not all) of the use cases for mobile apps.

9.6 Application Architectures

To handle their complexity, large applications are often broken into several layers:

- The *presentation* or *user-interface* layer, which deals with user interaction. A single application may have several different versions of this layer, corresponding to distinct kinds of interfaces such as web browsers and user interfaces of mobile phones, which have much smaller screens.

  In many implementations, the presentation/user-interface layer is itself conceptually broken up into layers, based on the **model-view-controller** (MVC) architecture. The **model** corresponds to the business-logic layer, described below. The **view** defines the presentation of data; a single underlying model can have different views depending on the specific software/device used to access the application. The **controller** receives events (user actions), executes actions on the model, and returns

Telegram: @uni_k

www.konkur.in

a view to the user. The MVC architecture is used in a number of web application frameworks.

- The **business-logic** layer, which provides a high-level view of data and actions on data. We discuss the business-logic layer in more detail in Section 9.6.1.
- The **data-access** layer, which provides the interface between the business-logic layer and the underlying database. Many applications use an object-oriented language to code the business-logic layer and use an object-oriented model of data, while the underlying database is a relational database. In such cases, the data-access layer also provides the mapping from the object-oriented data model used by the business logic to the relational model supported by the database. We discuss such mappings in more detail in Section 9.6.2.

Figure 9.14 shows these layers, along with a sequence of steps taken to process a request from the web browser. The labels on the arrows in the figure indicate the order of the steps. When the request is received by the application server, the controller sends a request to the model. The model processes the request, using business logic, which may involve updating objects that are part of the model, followed by creating a result object. The model in turn uses the data-access layer to update or retrieve information from a database. The result object created by the model is sent to the view module, which creates an HTML view of the result to be displayed on the web browser. The view may be tailored based on the characteristics of the device used to view the result —for example, whether it is a computer monitor with a large screen or a small screen on a phone. Increasingly, the view layer is implemented by code running at the client, instead of at the server.

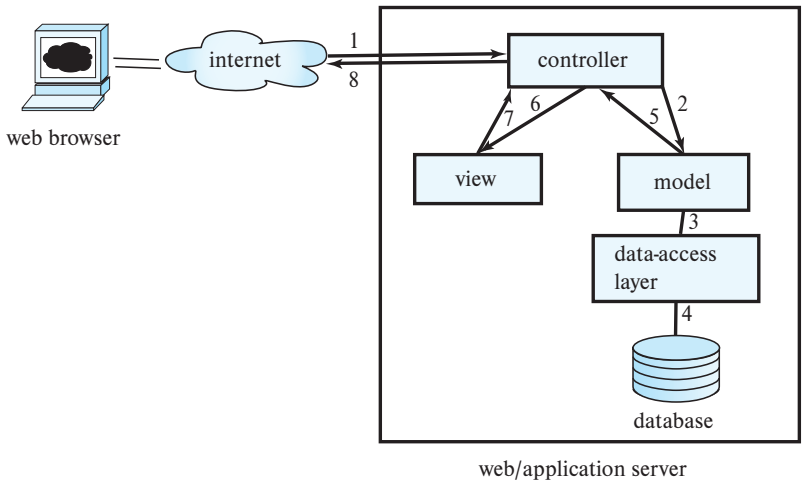


Figure 9.14 Web application architecture.

Telegram: @uni_k

www.konkur.in

9.6.1 The Business-Logic Layer

The business-logic layer of an application for managing a university may provide abstractions of entities such as students, instructors, courses, sections, etc., and actions such as admitting a student to the university, enrolling a student in a course, and so on. The code implementing these actions ensures that **business rules** are satisfied; for example, the code would ensure that a student can enroll for a course only if she has already completed course prerequisites and has paid her tuition fees.

In addition, the business logic includes **workflows**, which describe how a particular task that involves multiple participants is handled. For example, if a candidate applies to the university, there is a workflow that defines who should see and approve the application first, and if approved in the first step, who should see the application next, and so on until either an offer is made to the student, or a rejection note is sent out. Workflow management also needs to deal with error situations; for example, if a deadline for approval/rejection is not met, a supervisor may need to be informed so she can intervene and ensure the application is processed.

9.6.2 The Data-Access Layer and Object-Relational Mapping

In the simplest scenario, where the business-logic layer uses the same data model as the database, the data-access layer simply hides the details of interfacing with the database. However, when the business-logic layer is written using an object-oriented programming language, it is natural to model data as objects, with methods invoked on objects.

In early implementations, programmers had to write code for creating objects by fetching data from the database and for storing updated objects back in the database. However, such manual conversions between data models is cumbersome and error prone. One approach to handling this problem was to develop a database system that natively stores objects, and relationships between objects, and allows objects in the database to be accessed in exactly the same way as in-memory objects. Such databases, called **object-oriented databases**, were discussed in Section 8.2. However, object-oriented databases did not achieve commercial success for a variety of technical and commercial reasons.

An alternative approach is to use traditional relational databases to store data, but to automate the mapping of data in relation to in-memory objects, which are created on demand (since memory is usually not sufficient to store all data in the database), as well as the reverse mapping to store updated objects back as relations in the database.

Several systems have been developed to implement such **object-relational mappings**. We describe the Hibernate and Django ORMs next.

9.6.2.1 Hibernate ORM

The **Hibernate** system is widely used for mapping from Java objects to relations. Hibernate provides an implementation of the Java Persistence API (JPA). In Hibernate, the mapping from each Java class to one or more relations is specified in a mapping file.

Telegram: @uni_k

www.konkur.in

The mapping file can specify, for example, that a Java class called Student is mapped to the relation *student*, with the Java attribute ID mapped to the attribute *student*.ID, and so on. Information about the database, such as the host on which it is running and user name and password for connecting to the database, are specified in a *properties* file. The program has to open a *session*, which sets up the connection to the database. Once the session is set up, a Student object stud created in Java can be stored in the database by invoking session.save(stud). The Hibernate code generates the SQL commands required to store corresponding data in the *student* relation.

While entities in an E-R model naturally correspond to objects in an object-oriented language such as Java, relationships often do not. Hibernate supports the ability to map such relationships as sets associated with objects. For example, the *takes* relationship between *student* and *section* can be modeled by associating a set of *section*s with each *student*, and a set of *student*s with each *section*. Once the appropriate mapping is specified, Hibernate populates these sets automatically from the database relation *takes*, and updates to the sets are reflected back to the database relation on commit.

As an example of the use of Hibernate, we create a Java class corresponding to the *student* relation as follows:

```
@Entity public class Student {
        @Id String ID;
        String name;
        String department;
        int tot_cred;
}
```

To be precise, the class attributes should be declared as private, and getter/setter methods should be provided to access the attributes, but we omit these details.

The mapping of the class attributes of Student to attributes of the relation *student* can be specified in a mapping file, in an XML format, or more conveniently, by means of annotations of the Java code. In the example above, the annotation @Entity denotes that the class is mapped to a database relation, whose name by default is the class name, and whose attributes are by default the same as the class attributes. The default relation name and attribute names can be overridden using @Table and @Column annotations. The @Id annotation in the example specifies that ID is the primary key attribute.

The following code snippet then creates a Student object and saves it to the database.

```
Session session = getSessionFactory().openSession();
Transaction txn = session.beginTransaction();
Student stud = new Student("12328", "John Smith", "Comp. Sci.", 0);
session.save(stud);
txn.commit();
session.close();
```

Telegram: @uni_k

www.konkur.in

Hibernate automatically generates the required SQL **insert** statement to create a *student* tuple in the database.

Objects can be retrieved either by primary key or by a query, as illustrated in the following code snippet:

```
Session session = getSessionFactory().openSession();
Transaction txn = session.beginTransaction();
// Retrieve student object by identifier
Student stud1 = session.get(Student.class, "12328");
        .. print out the Student information ..
List students =
        session.createQuery("from Student as s order by s.ID asc").list();
for ( Iterator iter = students.iterator(); iter.hasNext(); ) {
        Student stud = (Student) iter.next();
        .. print out the Student information ..
}
txn.commit();
session.close();
```

A single object can be retrieved using the session.get() method by providing its class and its primary key. The retrieved object can be updated in memory; when the transaction on the ongoing Hibernate session is committed, Hibernate automatically saves the updated objects by making corresponding updates on relations in the database.

The preceding code snippet also shows a query in Hibernate's HQL query language, which is based on SQL but designed to allow objects to be used directly in the query. The HQL query is automatically translated to SQL by Hibernate and executed, and the results are converted into a list of Student objects. The for loop iterates over the objects in this list.

These features help to provide the programmer with a high-level model of data without bothering about the details of the relational storage. However, Hibernate, like other object-relational mapping systems, also allows queries to be written using SQL on the relations stored in the database; such direct database access, bypassing the object model, can be quite useful for writing complex queries.

9.6.2.2 The Django ORM

Several ORMs have been developed for the Python language. The ORM component of the Django framework is one of the most popular such ORMs, while SQLAlchemy is another popular Python ORM.

Figure 9.15 shows a model definition for Student and Instructor in Django. Observe that all of the fields of *student* and *instructor* have been defined as fields in the class Student and Instructor, with appropriate type definitions.

In addition, the relation *advisor* has been modeled here as a many-to-many relationship between Student and Instructor. The relationship is accessed by an attribute

Telegram: @uni_k

www.konkur.in


```
from django.db import models

class student(models.Model):
    id = models.CharField(primary_key=True, max_length=5)
    name = models.CharField(max_length=20)
    dept_name = models.CharField(max_length=20)
    tot_cred = models.DecimalField(max_digits=3, decimal_places=0)

class instructor(models.Model):
    id = models.CharField(primary_key=True, max_length=5)
    name = models.CharField(max_length=20)
    dept_name = models.CharField(max_length=20)
    salary = models.DecimalField(max_digits=8, decimal_places=2)
    advisees = models.ManyToManyField(student, related_name="advisors")
```

Figure 9.15 Model definition in Django.

called advisees in Instructor, which stores a set of references to Student objects. The reverse relationship from Student to Instructor is created automatically, and the model specifies that the reverse relationship attribute in the Student class is named advisors; this attribute stores a set of references to Instructor objects.

The Django view person_query_model shown in Figure 9.16 illustrates how to access database objects directly from the Python language, without using SQL. The expression Student.objects.filter() returns all student objects that satisfy the specified filter condition; in this case, students with the given name. The student names are printed out along with the names of their advisors. The expression Student.advisors.all() returns a list of advisors (advisor objects) of a given student, whose names are then retrieved and returned by the get_names() function. The case for instructors is similar, with instructor names being printed out along with the names of their advisees.

Django provides a tool called *migrate*, which creates database relations from a given model. Models can be given version numbers. When migrate is invoked on a model with a new version number, while an earlier version number is already in the database, the migrate tool also generates SQL code for migrating the existing data from the old database schema to the new database schema. It is also possible to create Django models from existing database schemas.

9.7 Application Performance

Web sites may be accessed by millions of people from across the globe, at rates of thousands of requests per second, or even more, for the most popular sites. Ensuring

Telegram: @uni_k

www.konkur.in


```
from models import Student, Instructor
def get_names(persons):
    res = ""
    for p in persons:
        res += p.name + ", "
    return res.rstrip(", ")

def person_query_model(request):
    persontype = request.GET.get('persontype')
    personname = request.GET.get('personname')
    html = ""

    if persontype == 'student':
        students = Student.objects.filter(name=personname)
        for student in students:
            advisors = students.advisors.all()
            html = html + "Advisee: " + student.name + "<br>Advisors: "
                + get_names(advisors) + "<br> \n"
    else:
        instructors = Instructor.objects.filter(name=personname)
        for instructor in instructors:
            advisees = instructor.advisees.all()
            html = html+"Advisor: " + instructor.name + "<br>Advisees: "
                + get_names(advisees) + "<br> \n"
    return HttpResponse(html)
```

Figure 9.16 View definition in Django using models.

that requests are served with low response times is a major challenge for web-site developers. To do so, application developers try to speed up the processing of individual requests by using techniques such as caching, and they exploit parallel processing by using multiple application servers. We describe these techniques briefly next. Tuning of database applications is another way to improve performance and is described in Section 25.1.

9.7.1 Reducing Overhead by Caching

Suppose that the application code for servicing each user request connects to a database through JDBC. Creating a new JDBC connection may take several milliseconds, so opening a new connection for each user request is not a good idea if very high transaction rates are to be supported.

Telegram: @uni_k

www.konkur.in

The **connection pooling** method is used to reduce this overhead; it works as follows: The connection pool manager (typically a part of the application server) creates a pool (that is, a set) of open ODBC/JDBC connections. Instead of opening a new connection to the database, the code servicing a user request (typically a servlet) asks for (requests) a connection from the connection pool and returns the connection to the pool when the code (servlet) completes its processing. If the pool has no unused connections when a connection is requested, a new connection is opened to the database (taking care not to exceed the maximum number of connections that the database system can support concurrently). If there are many open connections that have not been used for a while, the connection pool manager may close some of the open database connections. Many application servers and newer ODBC/JDBC drivers provide a built-in connection pool manager.

Details of how to create a connection pool vary by application server or JDBC driver, but most implementations require the creation of a `DataSource` object using the JDBC connection details such as the machine, port, database, user-id and password, as well as other parameters related to connection pooling. The `getConnection()` method invoked on the `DataSource` object gets a connection from the connection pool. Closing the connection returns the connection to the pool.

Certain requests may result in exactly the same query being resubmitted to the database. The cost of communication with the database can be greatly reduced by caching the results of earlier queries and reusing them, so long as the query result has not changed at the database. Some web servers support such query-result caching; caching can otherwise be done explicitly in application code.

Costs can be further reduced by caching the final web page that is sent in response to a request. If a new request comes with exactly the same parameters as a previous request, the request does not perform any updates, and the resultant web page is in the cache, that page can be reused to avoid the cost of recomputing the page. Caching can be done at the level of fragments of web pages, which are then assembled to create complete web pages.

Cached query results and cached web pages are forms of materialized views. If the underlying database data change, the cached results must be discarded, or recomputed, or even incrementally updated, as in materialized-view maintenance (described in Section 16.5). Some database systems (such as Microsoft SQL Server) provide a way for the application server to register a query with the database and get a **notification** from the database when the result of the query changes. Such a notification mechanism can be used to ensure that query results cached at the application server are up-to-date.

There are several widely used main-memory caching systems; among the more popular ones are **memcached** and **Redis**. Both systems allow applications to store data with an associated key and retrieve data for a specified key. Thus, they act as hash-map data structures that allow data to be stored in the main memory but also provide cache eviction of infrequently used data.

Telegram: @uni_k

www.konkur.in

For example, with memcached, data can be stored using memcached.add(key, data) and fetched using memcached.fetch(key). Instead of issuing a database query to fetch user data with a specified key, say key1, from a relation r, an application would first check if the required data are already cached by issuing a fetch("r:"+key1) (here, the key is appended to the relation name, to distinguish data from different relations that may be stored in the same memcached instance). If the fetch returns null, the database query is issued, a copy of the data fetched from the database is stored in memcached, and the data are then returned to the user. If the fetch does find the requested data, it can be used without accessing the database, leading to much faster access.

A client can connect to multiple memcached instances, which may run on different machines and store/retrieve data from any of them. How to decide what data are stored on which instance is left to the client code. By partitioning the data storage across multiple machines, an application can benefit from the aggregate main memory available across all the machines.

Memcached does not support automatic invalidation of cached data, but the application can track database changes and issue updates (using memcached_set(key, newvalue)) or deletes (using memcached_delete(key)) for the key values affected by update or deletion in the database. Redis offers very similar functionality. Both memcached and Redis provide APIs in multiple languages.

9.7.2 Parallel Processing

A commonly used approach to handling such very heavy loads is to use a large number of application servers running in parallel, each handling a fraction of the requests. A web server or a network router can be used to route each client request to one of the application servers. All requests from a particular client session must go to the same application server, since the server maintains state for a client session. This property can be ensured, for example, by routing all requests from a particular IP address to the same application server. The underlying database is, however, shared by all the application servers, so users see a consistent view of the database.

While the above architecture ensures that application servers do not become bottlenecks, it cannot prevent the database from becoming a bottleneck, since there is only one database server. To avoid overloading the database, application designers often use caching techniques to reduce the number of requests to the database. In addition, parallel database systems, described in Chapter 21 through Chapter 23, are used when the database needs to handle very large amounts of data, or a very large query load. Parallel data storage systems that are accessible via web service APIs are also popular in applications that need to scale to a very large number of users.

9.8 Application Security

Application security has to deal with several security threats and issues beyond those handled by SQL authorization.

Telegram: @uni_k

www.konkur.in

The first point where security has to be enforced is in the application. To do so, applications must authenticate users and ensure that users are only allowed to carry out authorized tasks.

There are many ways in which an application's security can be compromised, even if the database system is itself secure, due to badly written application code. In this section, we first describe several security loopholes that can permit hackers to carry out actions that bypass the authentication and authorization checks carried out by the application, and we explain how to prevent such loopholes. Later in the section, we describe techniques for secure authentication, and for fine-grained authorization. We then describe audit trails that can help in recovering from unauthorized access and from erroneous updates. We conclude the section by describing issues in data privacy.

9.8.1 SQL Injection

In **SQL injection** attacks, the attacker manages to get an application to execute an SQL query created by the attacker. In Section 5.1.1.5, we saw an example of an SQL injection vulnerability if user inputs are concatenated directly with an SQL query and submitted to the database. As another example of SQL injection vulnerability, consider the form source text shown in Figure 9.3. Suppose the corresponding servlet shown in Figure 9.7 creates an SQL query string using the following Java expression:

```
String query = "select * from student where name like '%"
                + name + "%' "
```

where name is a variable containing the string input by the user, and then executes the query on the database. A malicious attacker using the web form can then type a string such as " '; <some SQL statement>; -- ", where <some SQL statement> denotes any SQL statement that the attacker desires, in place of a valid student name. The servlet would then execute the following string.

```
select * from student where name like '%'; <some SQL statement>; -- %'
```

The quote inserted by the attacker closes the string, the following semicolon terminates the query, and the following text inserted by the attacker gets interpreted as a second SQL query, while the closing quote has been commented out. Thus, the malicious user has managed to insert an arbitrary SQL statement that is executed by the application. The statement can cause significant damage, since it can perform any action on the database, bypassing all security measures implemented in the application code.

As discussed in Section 5.1.1.5, to avoid such attacks, it is best to use prepared statements to execute SQL queries. When setting a parameter of a prepared query, JDBC automatically adds escape characters so that the user-supplied quote would no longer be able to terminate the string. Equivalently, a function that adds such escape

Telegram: @uni_k

www.konkur.in

characters could be applied on input strings before they are concatenated with the SQL query, instead of using prepared statements.

Another source of SQL-injection risk comes from applications that create queries dynamically, based on selection conditions and ordering attributes specified in a form. For example, an application may allow a user to specify what attribute should be used for sorting the results of a query. An appropriate SQL query is constructed, based on the attribute specified. Suppose the application takes the attribute name from a form, in the variable orderAttribute, and creates a query string such as the following:

```
String query = "select * from takes order by " + orderAttribute;
```

A malicious user can send an arbitrary string in place of a meaningful orderAttribute value, even if the HTML form used to get the input tried to restrict the allowed values by providing a menu. To avoid this kind of SQL injection, the application should ensure that the orderAttribute variable value is one of the allowed values (in our example, attribute names) before appending it.

9.8.2 Cross-Site Scripting and Request Forgery

A web site that allows users to enter text, such as a comment or a name, and then stores it and later displays it to other users, is potentially vulnerable to a kind of attack called a **cross-site scripting** (**XSS**) attack. In such an attack, a malicious user enters code written in a client-side scripting language such as JavaScript or Flash instead of entering a valid name or comment. When a different user views the entered text, the browser executes the script, which can carry out actions such as sending private cookie information back to the malicious user or even executing an action on a different web server that the user may be logged into.

For example, suppose the user happens to be logged into her bank account at the time the script executes. The script could send cookie information related to the bank account login back to the malicious user, who could use the information to connect to the bank's web server, fooling it into believing that the connection is from the original user. Or the script could access appropriate pages on the bank's web site, with appropriately set parameters, to execute a money transfer. In fact, this particular problem can occur even without scripting by simply using a line of code such as

```
<img src=
    "https://mybank.com/transfermoney?amount=1000&toaccount=14523">
```

assuming that the URL mybank.com/transfermoney accepts the specified parameters and carries out a money transfer. This latter kind of vulnerability is also called **cross-site request forgery** or **XSRF** (sometimes also called **CSRF**).

XSS can be done in other ways, such as luring a user into visiting a web site that has malicious scripts embedded in its pages. There are other more complex kinds of XSS

Telegram: @uni_k

www.konkur.in

and XSRF attacks, which we shall not get into here. To protect against such attacks, two things need to be done:

- **Prevent your web site from being used to launch XSS or XSRF attacks.**
  The simplest technique is to disallow any HTML tags whatsoever in text input by users. There are functions that detect or strip all such tags. These functions can be used to prevent HTML tags, and as a result, any scripts, from being displayed to other users. In some cases HTML formatting is useful, and in that case functions that parse the text and allow limited HTML constructs but disallow other dangerous constructs can be used instead; these must be designed carefully, since something as innocuous as an image include could potentially be dangerous in case there is a bug in the image display software that can be exploited.
- **Protect your web site from XSS or XSRF attacks launched from other sites.**
  If the user has logged into your web site and visits a different web site vulnerable to XSS, the malicious code executing on the user's browser could execute actions on your web site or pass session information related to your web site back to the malicious user, who could try to exploit it. This cannot be prevented altogether, but you can take a few steps to minimize the risk.
  - The HTTP protocol allows a server to check the **referer** of a page access, that is, the URL of the page that had the link that the user clicked on to initiate the page access. By checking that the referer is valid, for example, that the referer URL is a page on the same web site, XSS attacks that originated on a different web page accessed by the user can be prevented.
  - Instead of using only the cookie to identify a session, the session could also be restricted to the IP address from which it was originally authenticated. As a result, even if a malicious user gets a cookie, he may not be able to log in from a different computer.
  - Never use a `GET` method to perform any updates. This prevents attacks using `<img src ..>` such as the one we saw earlier. In fact, the HTTP standard specifies that `GET` methods should not perform any updates.
  - If you use a web application framework like Django, make sure to use the XSRF/CSRF protection mechanisms provided by the framework.

9.8.3 Password Leakage

Another problem that application developers must deal with is storing passwords in clear text in the application code. For example, programs such as JSP scripts often contain passwords in clear text. If such scripts are stored in a directory accessible by a web server, an external user may be able to access the source code of the script and get access to the password for the database account used by the application. To avoid such problems, many application servers provide mechanisms to store passwords in

Telegram: @uni_k

www.konkur.in

encrypted form, which the server decrypts before passing it on to the database. Such a feature removes the need for storing passwords as clear text in application programs. However, if the decryption key is also vulnerable to being exposed, this approach is not fully effective.

As another measure against compromised database passwords, many database systems allow access to the database to be restricted to a given set of internet addresses, typically, the machines running the application servers. Attempts to connect to the database from other internet addresses are rejected. Thus, unless the malicious user is able to log into the application server, she cannot do any damage even if she gains access to the database password.

9.8.4 Application-Level Authentication

Authentication refers to the task of verifying the identity of a person/software connecting to an application. The simplest form of authentication consists of a secret password that must be presented when a user connects to the application. Unfortunately, passwords are easily compromised, for example, by guessing, or by sniffing of packets on the network if the passwords are not sent encrypted. More robust schemes are needed for critical applications, such as online bank accounts. Encryption is the basis for more robust authentication schemes. Authentication through encryption is addressed in Section 9.9.3.

Many applications use **two-factor authentication**, where two independent *factors* (i.e., pieces of information or processes) are used to identify a user. The two factors should not share a common vulnerability; for example, if a system merely required two passwords, both could be vulnerable to leakage in the same manner (by network sniffing, or by a virus on the computer used by the user, for example). While biometrics such as fingerprints or iris scanners can be used in situations where a user is physically present at the point of authentication, they are not very meaningful across a network.

Passwords are used as the first factor in most such two-factor authentication schemes. Smart cards or other encryption devices connected through the USB interface, which can be used for authentication based on encryption techniques (see Section 9.9.3), are widely used as second factors.

One-time password devices, which generate a new pseudo-random number (say) every minute are also widely used as a second factor. Each user is given one of these devices and must enter the number displayed by the device at the time of authentication, along with the password, to authenticate himself. Each device generates a different sequence of pseudo-random numbers. The application server can generate the same sequence of pseudo-random numbers as the device given to the user, stopping at the number that would be displayed at the time of authentication, and verify that the numbers match. This scheme requires that the clock in the device and at the server are synchronized reasonably closely.

Yet another second-factor approach is to send an SMS with a (randomly generated) one-time password to the user's phone (whose number is registered earlier) whenever

Telegram: @uni_k

www.konkur.in

the user wishes to log in to the application. The user must possess a phone with that number to receive the SMS and then enter the one-time password, along with her regular password, to be authenticated.

It is worth noting that even with two-factor authentication, users may still be vulnerable to **man-in-the-middle attacks**. In such attacks, a user attempting to connect to the application is diverted to a fake web site, which accepts the password (including second factor passwords) from the user and uses it immediately to authenticate to the original application. The HTTPS protocol, described in Section 9.9.3.2, is used to authenticate the web site to the user (so the user does not connect to a fake site believing it to be the intended site). The HTTPS protocol also encrypts data and prevents man-in-the-middle attacks.

When users access multiple web sites, it is often annoying for a user to have to authenticate herself to each site separately, often with different passwords on each site. There are systems that allow the user to authenticate herself to one central authentication service, and other web sites and applications can authenticate the user through the central authentication service; the same password can then be used to access multiple sites. The LDAP protocol is widely used to implement such a central point of authentication for applications within a single organization; organizations implement an LDAP server containing user names and password information, and applications use the LDAP server to authenticate users.

In addition to authenticating users, a central authentication service can provide other services, for example, providing information about the user such as name, email, and address information, to the application. This obviates the need to enter this information separately in each application. LDAP can be used for this task, as described in Section 25.5.2. Other directory systems such Microsoft's Active Directories also provide mechanisms for authenticating users as well as for providing user information.

A **single sign-on** system further allows the user to be authenticated once, and multiple applications can then verify the user's identity through an authentication service without requiring reauthentication. In other words, once a user is logged in at one site, he does not have to enter his user name and password at other sites that use the same single sign-on service. Such single sign-on mechanisms have long been used in network authentication protocols such as Kerberos, and implementations are now available for web applications.

The **Security Assertion Markup Language** (**SAML**) is a protocol for exchanging authentication and authorization information between different security domains, to provide cross-organization single sign-on. For example, suppose an application needs to provide access to all students from a particular university, say Yale. The university can set up a web-based service that carries out authentication. Suppose a user connects to the application with a username such as "joe@yale.edu". The application, instead of directly authenticating a user, diverts the user to Yale University's authentication service, which authenticates the user and then tells the application who the user is and

Telegram: @uni_k

www.konkur.in

may provide some additional information such as the category of the user (student or instructor) or other relevant information. The user's password and other authentication factors are never revealed to the application, and the user need not register explicitly with the application. However, the application must trust the university's authentication service when authenticating a user.

The **OpenID** protocol is an alternative for single sign-on across organizations, which works in a manner similar to SAML. The **OAuth** protocol is another protocol that allows users to authorize access to certain resources, via sharing of an authorization token.

9.8.5 Application-Level Authorization

Although the SQL standard supports a fairly flexible system of authorization based on roles (described in Section 4.7), the SQL authorization model plays a very limited role in managing user authorizations in a typical application. For instance, suppose you want all students to be able to see their own grades, but not the grades of anyone else. Such authorization cannot be specified in SQL for at least two reasons:

1. **Lack of end-user information**. With the growth in the web, database accesses come primarily from web application servers. The end users typically do not have individual user identifiers on the database itself, and indeed there may only be a single user identifier in the database corresponding to all users of an application server. Thus, authorization specification in SQL cannot be used in the above scenario.

   It is possible for an application server to authenticate end users and then pass the authentication information on to the database. In this section we will assume that the function *syscontext.user_id*() returns the identifier of the application user on whose behalf a query is being executed. [6]

2. **Lack of fine-grained authorization**. Authorization must be at the level of individual tuples if we are to authorize students to see only their own grades. Such authorization is not possible in the current SQL standard, which permits authorization only on an entire relation or view, or on specified attributes of relations or views.

   We could try to get around this limitation by creating for each student a view on the *takes* relation that shows only that student's grades. While this would work in principle, it would be extremely cumbersome since we would have to create one such view for every single student enrolled in the university, which is completely impractical.[7]

   An alternative is to create a view of the form

[6] In Oracle, a JDBC connection using Oracle's JDBC drivers can set the end user identifier using the method `OracleConnection.setClientIdentifier(userId)`, and an SQL query can use the function `sys_context('USERENV', 'CLIENT_IDENTIFIER')` to retrieve the user identifier.

[7] Database systems are designed to manage large relations but to manage schema information such as views in a way that assumes smaller data volumes so as to enhance overall

Telegram: @uni_k

www.konkur.in


```
create view studentTakes as
select *
from takes
where takes.ID= syscontext.user_id()
```

Users are then given authorization to this view, rather than to the underlying *takes* relation. However, queries executed on behalf of students must now be written on the view *studentTakes*, rather than on the original *takes* relation, whereas queries executed on behalf of instructors may need to use a different view. The task of developing applications becomes more complex as a result.

The task of authorization is often typically carried out entirely in the application, bypassing the authorization facilities of SQL. At the application level, users are authorized to access specific interfaces, and they may further be restricted to view or update certain data items only.

While carrying out authorization in the application gives a great deal of flexibility to application developers, there are problems, too.

- The code for checking authorization becomes intermixed with the rest of the application code.
- Implementing authorization through application code, rather than specifying it declaratively in SQL, makes it hard to ensure the absence of loopholes. Because of an oversight, one of the application programs may not check for authorization, allowing unauthorized users access to confidential data.

Verifying that all application programs make all required authorization checks involves reading through all the application-server code, a formidable task in a large system. In other words, applications have a very large "surface area," making the task of protecting the application significantly harder. And in fact, security loopholes have been found in a variety of real-life applications.

In contrast, if a database directly supported fine-grained authorization, authorization policies could be specified and enforced at the SQL level, which has a much smaller surface area. Even if some of the application interfaces inadvertently omit required authorization checks, the SQL-level authorization could prevent unauthorized actions from being executed.

Some database systems provide mechanisms for row-level authorization as we saw in Section 4.7.7. For example, the Oracle **Virtual Private Database** (**VPD**) allows a system administrator to associate a function with a relation; the function returns a predicate that must be added to any query that uses the relation (different functions can be defined for relations that are being updated). For example, using our syntax for retrieving application user identifiers, the function for the *takes* relation can return a predicate such as:

Telegram: @uni_k

www.konkur.in

$$ID = sys_context.user_id()$$

This predicate is added to the **where** clause of every query that uses the *takes* relation. As a result (assuming that the application program sets the *user_id* value to the student's *ID*), each student can see only the tuples corresponding to courses that she took.

As we discussed in Section 4.7.7, a potential pitfall with adding a predicate as described above is that it may change the meaning of a query. For example, if a user wrote a query to find the average grade over all courses, she would end up getting the average of *her* grades, not all grades. Although the system would give the "right" answer for the rewritten query, that answer would not correspond to the query the user may have thought she was submitting.

PostgreSQL and Microsoft SQL Server offer row-level authorization support with similar functionality to Oracle VPD. More information on Oracle VPD and PostgreSQL and SQL Server row-level authorization may be found in their respective system manuals available online.

9.8.6 Audit Trails

An **audit trail** is a log of all changes (inserts, deletes, and updates) to the application data, along with information such as which user performed the change and when the change was performed. If application security is breached, or even if security was not breached, but some update was carried out erroneously, an audit trail can (a) help find out what happened, and who may have carried out the actions, and (b) aid in fixing the damage caused by the security breach or erroneous update.

For example, if a student's grade is found to be incorrect, the audit log can be examined to locate when and how the grade was updated, as well as to find which user carried out the updates. The university could then also use the audit trail to trace all the updates performed by this user in order to find other incorrect or fraudulent updates, and then correct them.

Audit trails can also be used to detect security breaches where a user's account is compromised and accessed by an intruder. For example, each time a user logs in, she may be informed about all updates in the audit trail that were done from that login in the recent past; if the user sees an update that she did not carry out, it is likely the account has been compromised.

It is possible to create a database-level audit trail by defining appropriate triggers on relation updates (using system-defined variables that identify the user name and time). However, many database systems provide built-in mechanisms to create audit trails that are much more convenient to use. Details of how to create audit trails vary across database systems, and you should refer to the database-system manuals for details.

Database-level audit trails are usually insufficient for applications, since they are usually unable to track who was the end user of the application. Further, updates are recorded at a low level, in terms of updates to tuples of a relation, rather than at a higher level, in terms of the business logic. Applications, therefore, usually create a

Telegram: @uni_k

www.konkur.in

higher-level audit trail, recording, for example, what action was carried out, by whom, when, and from which IP address the request originated.

A related issue is that of protecting the audit trail itself from being modified or deleted by users who breach application security. One possible solution is to copy the audit trail to a different machine, to which the intruder would not have access, with each record in the trail copied as soon as it is generated. A more robust solution is to use blockchain techniques, which are described in Chapter 26; blockchain techniques store logs in multiple machines and use a hashing mechanism that makes it very difficult for an intruder to modify or delete data without being detected.

9.8.7 Privacy

In a world where an increasing amount of personal data are available online, people are increasingly worried about the privacy of their data. For example, most people would want their personal medical data to be kept private and not revealed publicly. However, the medical data must be made available to doctors and emergency medical technicians who treat the patient. Many countries have laws on privacy of such data that define when and to whom the data may be revealed. Violation of privacy law can result in criminal penalties in some countries. Applications that access such private data must be built carefully, keeping the privacy laws in mind.

On the other hand, aggregated private data can play an important role in many tasks such as detecting drug side effects, or in detecting the spread of epidemics. How to make such data available to researchers carrying out such tasks without compromising the privacy of individuals is an important real-world problem. As an example, suppose a hospital hides the name of the patient but provides a researcher with the date of birth and the postal code of the patient (both of which may be useful to the researcher). Just these two pieces of information can be used to uniquely identify the patient in many cases (using information from an external database), compromising his privacy. In this particular situation, one solution would be to give the year of birth but not the date of birth, along with the address, which may suffice for the researcher. This would not provide enough information to uniquely identify most individuals.[8]

As another example, web sites often collect personal data such as address, telephone, email, and credit-card information. Such information may be required to carry out a transaction such as purchasing an item from a store. However, the customer may not want the information to be made available to other organizations, or may want part of the information (such as credit-card numbers) to be erased after some period of time as a way to prevent it from falling into unauthorized hands in the event of a security breach. Many web sites allow customers to specify their privacy preferences, and those web sites must then ensure that these preferences are respected.

[8]For extremely old people, who are relatively rare, even the year of birth plus postal code may be enough to uniquely identify the individual, so a range of values, such as 90 years or older, may be provided instead of the actual age for people older than 90 years.

Telegram: @uni_k

www.konkur.in

9.9 Encryption and Its Applications

Encryption refers to the process of transforming data into a form that is unreadable, unless the reverse process of decryption is applied. Encryption algorithms use an encryption key to perform encryption, and they require a decryption key (which could be the same as the encryption key, depending on the encryption algorithm used) to perform decryption.

The oldest uses of encryption were for transmitting messages, encrypted using a secret key known only to the sender and the intended receiver. Even if the message is intercepted by an enemy, the enemy, not knowing the key, will not be able to decrypt and understand the message. Encryption is widely used today for protecting data in transit in a variety of applications such as data transfer on the internet, and on cell-phone networks. Encryption is also used to carry out other tasks, such as authentication, as we will see in Section 9.9.3.

In the context of databases, encryption is used to store data in a secure way, so that even if the data are acquired by an unauthorized user (e.g., a laptop computer containing the data is stolen), the data will not be accessible without a decryption key.

Many databases today store sensitive customer information, such as credit-card numbers, names, fingerprints, signatures, and identification numbers such as, in the United States, social security numbers. A criminal who gets access to such data can use them for a variety of illegal activities, such as purchasing goods using a credit-card number, or even acquiring a credit card in someone else's name. Organizations such as credit-card companies use knowledge of personal information as a way of identifying who is requesting a service or goods. Leakage of such personal information allows a criminal to impersonate someone else and get access to service or goods; such impersonation is referred to as **identity theft**. Thus, applications that store such sensitive data must take great care to protect them from theft.

To reduce the chance of sensitive information being acquired by criminals, many countries and states today require by law that any database storing such sensitive information must store the information in an encrypted form. A business that does not protect its data thus could be held criminally liable in case of data theft. Thus, encryption is a critical component of any application that stores such sensitive information.

9.9.1 Encryption Techniques

There are a vast number of techniques for the encryption of data. Simple encryption techniques may not provide adequate security, since it may be easy for an unauthorized user to break the code. As an example of a weak encryption technique, consider the substitution of each character with the next character in the alphabet. Thus,

Perryridge

becomes

Qfsszsjehf

Telegram: @uni_k

www.konkur.in

If an unauthorized user sees only "Qfsszsjehf," she probably has insufficient information to break the code. However, if the intruder sees a large number of encrypted branch names, she could use statistical data regarding the relative frequency of characters to guess what substitution is being made (for example, *E* is the most common letter in English text, followed by *T, A, O, N, I*, and so on).

A good encryption technique has the following properties:

- It is relatively simple for authorized users to encrypt and decrypt data.
- It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the *encryption key*, which is used to encrypt data. In a **symmetric-key** encryption technique, the encryption key is also used to decrypt data. In contrast, in **public-key** (also known as **asymmetric-key**) encryption techniques, there are two different keys, the public key and the private key, used to encrypt and decrypt the data.
- Its decryption key is extremely difficult for an intruder to determine, even if the intruder has access to encrypted data. In the case of asymmetric-key encryption, it is extremely difficult to infer the private key even if the public key is available.

The **Advanced Encryption Standard** (**AES**) is a symmetric-key encryption algorithm that was adopted as an encryption standard by the U.S. government in 2000 and is now widely used. The standard is based on the **Rijndael algorithm** (named for the inventors V. Rijmen and J. Daemen). The algorithm operates on a 128-bit block of data at a time, while the key can be 128, 192, or 256 bits in length. The algorithm runs a series of steps to jumble up the bits in a data block in a way that can be reversed during decryption, and it performs an XOR operation with a 128-bit "round key" that is derived from the encryption key. A new round key is generated from the encryption key for each block of data that is encrypted. During decryption, the round keys are generated again from the encryption key and the encryption process is reversed to recover the original data. An earlier standard called the *Data Encryption Standard* (DES), adopted in 1977, was very widely used earlier.

For any symmetric-key encryption scheme to work, authorized users must be provided with the encryption key via a secure mechanism. This requirement is a major weakness, since the scheme is no more secure than the security of the mechanism by which the encryption key is transmitted.

Public-key encryption is an alternative scheme that avoids some of the problems faced by symmetric-key encryption techniques. It is based on two keys: a *public key* and a *private key*. Each user U_i has a public key E_i and a private key D_i. All public keys are published: They can be seen by anyone. Each private key is known to only the one user to whom the key belongs. If user U_1 wants to store encrypted data, U_1 encrypts them using public key E_1. Decryption requires the private key D_1.

Because the encryption key for each user is public, it is possible to exchange information securely by this scheme. If user U_1 wants to share data with U , U encrypts

Telegram: @uni_k

www.konkur.in

the data using E_2, the public key of U_2. Since only user U_2 knows how to decrypt the data, information can be transferred securely.

For public-key encryption to work, there must be a scheme for encryption such that it is infeasible (that is, extremely hard) to deduce the private key, given the public key. Such a scheme does exist and is based on these conditions:

- There is an efficient algorithm for testing whether or not a number is prime.
- No efficient algorithm is known for finding the prime factors of a number.

For purposes of this scheme, data are treated as a collection of integers. We create a public key by computing the product of two large prime numbers: P_1 and P_2. The private key consists of the pair (P_1, P_2). The decryption algorithm cannot be used successfully if only the product P_1P_2 is known; it needs the individual values P_1 and P_2. Since all that is published is the product P_1P_2, an unauthorized user would need to be able to factor P_1P_2 to steal data. By choosing P_1 and P_2 to be sufficiently large (over 100 digits), we can make the cost of factoring P_1P_2 prohibitively high (on the order of years of computation time, on even the fastest computers).

The details of public-key encryption and the mathematical justification of this technique's properties are referenced in the bibliographical notes.

Although public-key encryption by this scheme is secure, it is also computationally very expensive. A hybrid scheme widely used for secure communication is as follows: a symmetric encryption key (based, for example, on AES) is randomly generated and exchanged in a secure manner using a public-key encryption scheme, and symmetric-key encryption using that key is used on the data transmitted subsequently.

Encryption of small values, such as identifiers or names, is made complicated by the possibility of **dictionary attacks**, particularly if the encryption key is publicly available. For example, if date-of-birth fields are encrypted, an attacker trying to decrypt a particular encrypted value e could try encrypting every possible date of birth until he finds one whose encrypted value matches e. Even if the encryption key is not publicly available, statistical information about data distributions can be used to figure out what an encrypted value represents in some cases, such as age or address. For example, if the age 18 is the most common age in a database, the encrypted age value that occurs most often can be inferred to represent 18.

Dictionary attacks can be deterred by adding extra random bits to the end of the value before encryption (and removing them after decryption). Such extra bits, referred to as an **initialization vector** in AES, or as *salt* bits in other contexts, provide good protection against dictionary attack.

9.9.2 Encryption Support in Databases

Many file systems and database systems today support encryption of data. Such encryption protects the data from someone who is able to access the data but is not able to access the decryption key. In the case of file-system encryption, the data to be encrypted are usually large files and directories containing information about files.

Telegram: @uni_k

www.konkur.in

In the context of databases, encryption can be done at several different levels. At the lowest level, the disk blocks containing database data can be encrypted, using a key available to the database-system software. When a block is retrieved from disk, it is first decrypted and then used in the usual fashion. Such disk-block-level encryption protects against attackers who can access the disk contents but do not have access to the encryption key.

At the next higher level, specified (or all) attributes of a relation can be stored in encrypted form. In this case, each attribute of a relation could have a different encryption key. Many databases today support encryption at the level of specified attributes as well as at the level of an entire relation, or all relations in a database. Encryption of specified attributes minimizes the overhead of decryption by allowing applications to encrypt only attributes that contain sensitive values such as credit-card numbers. Encryption also then needs to use extra random bits to prevent dictionary attacks, as described earlier. However, databases typically do not allow primary and foreign key attributes to be encrypted, and they do not support indexing on encrypted attributes.

A decryption key is obviously required to get access to encrypted data. A single master encryption key may be used for all the encrypted data; with attribute level encryption, different encryption keys could be used for different attributes. In this case, the decryption keys for different attributes can be stored in a file or relation (often referred to as "wallet"), which is itself encrypted using a master key.

A connection to the database that needs to access encrypted attributes must then provide the master key; unless this is provided, the connection will not be able to access encrypted data. The master key would be stored in the application program (typically on a different computer), or memorized by the database user, and provided when the user connects to the database.

Encryption at the database level has the advantage of requiring relatively low time and space overhead and does not require modification of applications. For example, if data in a laptop computer database need to be protected from theft of the computer itself, such encryption can be used. Similarly, someone who gets access to backup tapes of a database would not be able to access the data contained in the backups without knowing the decryption key.

An alternative to performing encryption in the database is to perform it *before* the data are sent to the database. The application must then encrypt the data before sending it to the database and decrypt the data when they are retrieved. This approach to data encryption requires significant modifications to be done to the application, unlike encryption performed in a database system.

9.9.3 Encryption and Authentication

Password-based authentication is used widely by operating systems as well as database systems. However, the use of passwords has some drawbacks, especially over a network. If an eavesdropper is able to "sniff" the data being sent over the network, she may be able to find the password as it is being sent across the network. Once the eavesdropper

Telegram: @uni_k

www.konkur.in

has a user name and password, she can connect to the database, pretending to be the legitimate user.

A more secure scheme involves a **challenge-response** system. The database system sends a challenge string to the user. The user encrypts the challenge string using a secret password as encryption key and then returns the result. The database system can verify the authenticity of the user by decrypting the string with the same secret password and checking the result with the original challenge string. This scheme ensures that no passwords travel across the network.

Public-key systems can be used for encryption in challenge – response systems. The database system encrypts a challenge string using the user's public key and sends it to the user. The user decrypts the string using her private key and returns the result to the database system. The database system then checks the response. This scheme has the added benefit of not storing the secret password in the database, where it could potentially be seen by system administrators.

Storing the private key of a user on a computer (even a personal computer) has the risk that if the computer is compromised, the key may be revealed to an attacker who can then masquerade as the user. **Smart cards** provide a solution to this problem. In a smart card, the key can be stored on an embedded chip; the operating system of the smart card guarantees that the key can never be read, but it allows data to be sent to the card for encryption or decryption, using the private key.[9]

9.9.3.1 Digital Signatures

Another interesting application of public-key encryption is in **digital signatures** to verify authenticity of data; digital signatures play the electronic role of physical signatures on documents. The private key is used to "sign," that is, encrypt, data, and the signed data can be made public. Anyone can verify the signature by decrypting the data using the public key, but no one could have generated the signed data without having the private key. (Note the reversal of the roles of the public and private keys in this scheme.) Thus, we can **authenticate** the data; that is, we can verify that the data were indeed created by the person who is supposed to have created them.

Furthermore, digital signatures also serve to ensure **nonrepudiation**. That is, in case the person who created the data later claims she did not create them (the electronic equivalent of claiming not to have signed the check), we can prove that that person must have created the data (unless her private key was leaked to others).

9.9.3.2 Digital Certificates

Authentication is, in general, a two-way process, where each of a pair of interacting entities authenticates itself to the other. Such pairwise authentication is needed even

[9] Smart cards provide other functionality too, such as the ability to store cash digitally and make payments, which is not relevant in our context.

Telegram: @uni_k

www.konkur.in

when a client contacts a web site, to prevent a malicious site from masquerading as a legal web site. Such masquerading could be done, for example, if the network routers were compromised and data rerouted to the malicious site.

For a user to ensure that she is interacting with an authentic web site, she must have the site's public key. This raises the problem of how the user can get the public key—if it is stored on the web site, the malicious site could supply a different key, and the user would have no way of verifying if the supplied public key is itself authentic. Authentication can be handled by a system of **digital certificates**, whereby public keys are signed by a certification agency, whose public key is well known. For example, the public keys of the root certification authorities are stored in standard web browsers. A certificate issued by them can be verified by using the stored public keys.

A two-level system would place an excessive burden of creating certificates on the root certification authorities, so a multilevel system is used instead, with one or more root certification authorities and a tree of certification authorities below each root. Each authority (other than the root authorities) has a digital certificate issued by its parent.

A digital certificate issued by a certification authority A consists of a public key K_A and an encrypted text E that can be decoded by using the public key K_A. The encrypted text contains the name of the party to whom the certificate was issued and her public key K_c. In case the certification authority A is not a root certification authority, the encrypted text also contains the digital certificate issued to A by its parent certification authority; this certificate authenticates the key K_A itself. (That certificate may in turn contain a certificate from a further parent authority, and so on.)

To verify a certificate, the encrypted text E is decrypted by using the public key K_A to retrieve the name of the party (i.e., the name of the organization owning the web site); additionally, if A is not a root authority whose public key is known to the verifier, the public key K_A is verified recursively by using the digital certificate contained within E; recursion terminates when a certificate issued by the root authority is reached. Verifying the certificate establishes the chain through which a particular site was authenticated and provides the name and authenticated public key for the site.

Digital certificates are widely used to authenticate web sites to users, to prevent malicious sites from masquerading as other web sites. In the HTTPS protocol (the secure version of the HTTP protocol), the site provides its digital certificate to the browser, which then displays it to the user. If the user accepts the certificate, the browser then uses the provided public key to encrypt data. A malicious site will have access to the certificate, but not the private key, and will thus not be able to decrypt the data sent by the browser. Only the authentic site, which has the corresponding private key, can decrypt the data sent by the browser. We note that public-/private-key encryption and decryption costs are much higher than encryption/decryption costs using symmetric private keys. To reduce encryption costs, HTTPS actually creates a one-time symmetric key after authentication and uses it to encrypt data for the rest of the session.

Telegram: @uni_k

www.konkur.in

Digital certificates can also be used for authenticating users. The user must submit a digital certificate containing her public key to a site, which verifies that the certificate has been signed by a trusted authority. The user's public key can then be used in a challenge-response system to ensure that the user possesses the corresponding private key, thereby authenticating the user.

9.10 Summary

- Application programs that use databases as back ends and interact with users have been around since the 1960s. Application architectures have evolved over this period. Today most applications use web browsers as their front end, and a database as their back end, with an application server in between.
- HTML provides the ability to define interfaces that combine hyperlinks with forms facilities. Web browsers communicate with web servers by the HTTP protocol. Web servers can pass on requests to application programs and return the results to the browser.
- Web servers execute application programs to implement desired functionality. Servlets are a widely used mechanism to write application programs that run as part of the web server process, in order to reduce overhead. There are also many server-side scripting languages that are interpreted by the web server and provide application-program functionality as part of the web server.
- There are several client-side scripting languages—JavaScript is the most widely used—that provide richer user interaction at the browser end.
- Complex applications usually have a multilayer architecture, including a model implementing business logic, a controller, and a view mechanism to display results. They may also include a data access layer that implements an object-relational mapping. Many applications implement and use web services, allowing functions to be invoked over HTTP.
- Techniques such as caching of various forms, including query result caching and connection pooling, and parallel processing are used to improve application performance.
- Application developers must pay careful attention to security, to prevent attacks such as SQL injection attacks and cross-site scripting attacks.
- SQL authorization mechanisms are coarse grained and of limited value to applications that deal with large numbers of users. Today application programs implement fine-grained, tuple-level authorization, dealing with a large number of application users, completely outside the database system. Database extensions to provide tuple-level access control and to deal with large numbers of application users have been developed, but are not standard as yet.

Telegram: @uni_k

www.konkur.in

- Protecting the privacy of data are an important task for database applications. Many countries have legal requirements on protection of certain kinds of data, such as credit-card information or medical data.
- Encryption plays a key role in protecting information and in authentication of users and web sites. Symmetric-key encryption and public-key encryption are two contrasting but widely used approaches to encryption. Encryption of certain sensitive data stored in databases is a legal requirement in many countries and states.
- Encryption also plays a key role in authentication of users to applications, of Web sites to users, and for digital signatures.

Review Terms

- Application programs
- Web interfaces to databases
- HTML
- Hyperlinks
- Uniform resource locator (URL)
- Forms
- HyperText Transfer Protocol (HTTP)
- Connectionless protocols
- Cookie
- Session
- Servlets and Servlet sessions
- Server-side scripting
- Java Server Pages (JSP)
- PHP
- Client-side scripting
- JavaScript
- Document Object Model (DOM)
- Ajax
- Progressive Web Apps
- Application architecture
- Presentation layer
- Model-view-controller (MVC) architecture
- Business-logic layer
- Data-access layer
- Object-relational mapping
- Hibernate
- Django
- Web services
- RESTful web services
- Web application frameworks
- Connection pooling
- Query result caching
- Application security
- SQL injection
- Cross-site scripting (XSS)
- Cross-site request forgery (XSRF)
- Authentication
- Two-factor authentication
- Man-in-the-middle attack
- Central authentication
- Single sign-on
- OpenID
- Authorization
- Virtual Private Database (VPD)
- Audit trail

Telegram: @uni_k

www.konkur.in

- Encryption
- Symmetric-key encryption
- Public-key encryption
- Dictionary attack
- Challenge – response
- Digital signatures
- Digital certificates

Practice Exercises

9.1 What is the main reason why servlets give better performance than programs that use the common gateway interface (CGI), even though Java programs generally run slower than C or C++ programs?

9.2 List some benefits and drawbacks of connectionless protocols over protocols that maintain connections.

9.3 Consider a carelessly written web application for an online-shopping site, which stores the price of each item as a hidden form variable in the web page sent to the customer; when the customer submits the form, the information from the hidden form variable is used to compute the bill for the customer. What is the loophole in this scheme? (There was a real instance where the loophole was exploited by some customers of an online-shopping site before the problem was detected and fixed.)

9.4 Consider another carelessly written web application which uses a servlet that checks if there was an active session but does not check if the user is authorized to access that page, instead depending on the fact that a link to the page is shown only to authorized users. What is the risk with this scheme? (There was a real instance where applicants to a college admissions site could, after logging into the web site, exploit this loophole and view information they were not authorized to see; the unauthorized access was, however, detected, and those who accessed the information were punished by being denied admission.)

9.5 Why is it important to open JDBC connections using the try-with-resources (`try (…){ … }`) syntax?

9.6 List three ways in which caching can be used to speed up web server performance.

9.7 The `netstat` command (available on Linux and on Windows) shows the active network connections on a computer. Explain how this command can be used to find out if a particular web page is not closing connections that it opened, or if connection pooling is used, not returning connections to the connection pool. You should account for the fact that with connection pooling, the connection may not get closed immediately.

Telegram: @uni_k

www.konkur.in

9.8 Testing for SQL-injection vulnerability:

a. Suggest an approach for testing an application to find if it is vulnerable to SQL injection attacks on text input.

b. Can SQL injection occur with forms of HTML input other than text boxes? If so, how would you test for vulnerability?

9.9 A database relation may have the values of certain attributes encrypted for security. Why do database systems not support indexing on encrypted attributes? Using your answer to this question, explain why database systems do not allow encryption of primary-key attributes.

9.10 Exercise 9.9 addresses the problem of encryption of certain attributes. However, some database systems support encryption of entire databases. Explain how the problems raised in Exercise 9.9 are avoided if the entire database is encrypted.

9.11 Suppose someone impersonates a company and gets a certificate from a certificate-issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?

9.12 Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.

Exercises

9.13 Write a servlet and associated HTML code for the following very simple application: A user is allowed to submit a form containing a value, say n, and should get a response containing n "*" symbols.

9.14 Write a servlet and associated HTML code for the following simple application: A user is allowed to submit a form containing a number, say n, and should get a response saying how many times the value n has been submitted previously. The number of times each value has been submitted previously should be stored in a database.

9.15 Write a servlet that authenticates a user (based on user names and passwords stored in a database relation) and sets a session variable called *userid* after authentication.

9.16 What is an SQL injection attack? Explain how it works and what precautions must be taken to prevent SQL injection attacks.

9.17 Write pseudocode to manage a connection pool. Your pseudocode must include a function to create a pool (providing a database connection string, database user name, and password as parameters), a function to request a connection

Telegram: @uni_k

www.konkur.in

from the pool, a connection to release a connection to the pool, and a function to close the connection pool.

9.18 Explain the terms CRUD and REST.

9.19 Many web sites today provide rich user interfaces using Ajax. List two features each of which reveals if a site uses Ajax, without having to look at the source code. Using the above features, find three sites which use Ajax; you can view the HTML source of the page to check if the site is actually using Ajax.

9.20 XSS attacks:

a. What is an XSS attack?

b. How can the referer field be used to detect some XSS attacks?

9.21 What is multifactor authentication? How does it help safeguard against stolen passwords?

9.22 Consider the Oracle Virtual Private Database (VPD) feature described in Section 9.8.5 and an application based on our university schema.

a. What predicate (using a subquery) should be generated to allow each faculty member to see only *takes* tuples corresponding to course sections that they have taught?

b. Give an SQL query such that the query with the predicate added gives a result that is a subset of the original query result without the added predicate.

c. Give an SQL query such that the query with the predicate added gives a result containing a tuple that is not in the result of the original query without the added predicate.

9.23 What are two advantages of encrypting data stored in the database?

9.24 Suppose you wish to create an audit trail of changes to the *takes* relation.

a. Define triggers to create an audit trail, logging the information into a relation called, for example, *takes_trail*. The logged information should include the user-id (assume a function *user_id*() provides this information) and a timestamp, in addition to old and new values. You must also provide the schema of the *takes_trail* relation.

b. Can the preceding implementation guarantee that updates made by a malicious database administrator (or someone who manages to get the administrator's password) will be in the audit trail? Explain your answer.

9.25 Hackers may be able to fool you into believing that their web site is actually a web site (such as a bank or credit card web site) that you trust. This may be done by misleading email, or even by breaking into the network infrastructure

Telegram: @uni_k

www.konkur.in

and rerouting network traffic destined for, say mybank.com, to the hacker's site. If you enter your user name and password on the hacker's site, the site can record it and use it later to break into your account at the real site. When you use a URL such as https://mybank.com, the HTTPS protocol is used to prevent such attacks. Explain how the protocol might use digital certificates to verify authenticity of the site.

9.26 Explain what is a challenge – response system for authentication. Why is it more secure than a traditional password-based system?

Project Suggestions

Each of the following is a large project, which can be a semester-long project done by a group of students. The difficulty of the project can be adjusted easily by adding or deleting features.

You can choose to use either a web front-end using HTML5, or a mobile front-end on Android or iOS for your project.

Project 9.1 Pick your favorite interactive web site, such as Bebo, Blogger, Facebook, Flickr, Last.FM, Twitter, Wikipedia; these are just a few examples, there are many more. Most of these sites manage a large amount of data and use databases to store and process the data. Implement a subset of the functionality of the web site you picked. Implementing even a significant subset of the features of such a site is well beyond a course project, but it is possible to find a set of features that is interesting to implement yet small enough for a course project.

Most of today's popular web sites make extensive use of Javascript to create rich interfaces. You may wish to go easy on this for your project, at least initially, since it takes time to build such interfaces, and then add more features to your interfaces, as time permits.

Make use of web application development frameworks, or Javascript libraries available on the web, such as the jQuery library, to speed up your front-end development. Alternatively, implement the application as a mobile app on Android or iOS.

Project 9.2 Create a "mashup" which uses web services such as Google or Yahoo map APIs to create an interactive web site. For example, the map APIs provide a way to display a map on the web page, with other information overlaid on the maps. You could implement a restaurant recommendation system, with users contributing information about restaurants such as location, cuisine, price range, and ratings. Results of user searches could be displayed on the map. You could allow Wikipedia-like features, such as allowing users to add information and edit

Telegram: @uni_k

www.konkur.in

information added by other users, along with moderators who can weed out malicious updates. You could also implement social features, such as giving more importance to ratings provided by your friends.

Project 9.3 Your university probably uses a course-management system such as Moodle, Blackboard, or WebCT. Implement a subset of the functionality of such a course-management system. For example, you can provide assignment submission and grading functionality, including mechanisms for students and teachers/teaching assistants to discuss grading of a particular assignment. You could also provide polls and other mechanisms for getting feedback.

Project 9.4 Consider the E-R schema of Practice Exercise 6.3 (Chapter 6), which represents information about teams in a league. Design and implement a web-based system to enter, update, and view the data.

Project 9.5 Design and implement a shopping cart system that lets shoppers collect items into a shopping cart (you can decide what information is to be supplied for each item) and purchased together. You can extend and use the E-R schema of Exercise 6.21 of Chapter 6. You should check for availability of the item and deal with nonavailable items as you feel appropriate.

Project 9.6 Design and implement a web-based system to record student registration and grade information for courses at a university.

Project 9.7 Design and implement a system that permits recording of course performance information—specifically, the marks given to each student in each assignment or exam of a course, and computation of a (weighted) sum of marks to get the total course marks. The number of assignments/exams should not be predefined; that is, more assignments/exams can be added at any time. The system should also support grading, permitting cutoffs to be specified for various grades.

You may also wish to integrate it with the student registration system of Project 9.6 (perhaps being implemented by another project team).

Project 9.8 Design and implement a web-based system for booking classrooms at your university. Periodic booking (fixed days/times each week for a whole semester) must be supported. Cancellation of specific lectures in a periodic booking should also be supported.

You may also wish to integrate it with the student registration system of Project 9.6 (perhaps being implemented by another project team) so that classrooms can be booked for courses, and cancellations of a lecture or addition of extra lectures can be noted at a single interface and will be reflected in the classroom booking and communicated to students via email.

Telegram: @uni_k

www.konkur.in

Project 9.9 Design and implement a system for managing online multiple-choice tests. You should support distributed contribution of questions (by teaching assistants, for example), editing of questions by whoever is in charge of the course, and creation of tests from the available set of questions. You should also be able to administer tests online, either at a fixed time for all students or at any time but with a time limit from start to finish (support one or both), and the system should give students feedback on their scores at the end of the allotted time.

Project 9.10 Design and implement a system for managing email customer service. Incoming mail goes to a common pool. There is a set of customer service agents who reply to email. If the email is part of an ongoing series of replies (tracked using the in-reply-to field of email) the mail should preferably be replied to by the same agent who replied earlier. The system should track all incoming mail and replies, so an agent can see the history of questions from a customer before replying to an email.

Project 9.11 Design and implement a simple electronic marketplace where items can be listed for sale or for purchase under various categories (which should form a hierarchy). You may also wish to support alerting services, whereby a user can register interest in items in a particular category, perhaps with other constraints as well, without publicly advertising her interest, and is notified when such an item is listed for sale.

Project 9.12 Design and implement a web-based system for managing a sports "ladder." Many people register and may be given some initial rankings (perhaps based on past performance). Anyone can challenge anyone else to a match, and the rankings are adjusted according to the result. One simple system for adjusting rankings just moves the winner ahead of the loser in the rank order, in case the winner was behind earlier. You can try to invent more complicated rank-adjustment systems.

Project 9.13 Design and implement a publication-listing service. The service should permit entering of information about publications, such as title, authors, year, where the publication appeared, and pages. Authors should be a separate entity with attributes such as name, institution, department, email, address, and home page.

Your application should support multiple views on the same data. For instance, you should provide all publications by a given author (sorted by year, for example), or all publications by authors from a given institution or department. You should also support search by keywords, on the overall database as well as within each of the views.

Project 9.14 A common task in any organization is to collect structured information from a group of people. For example, a manager may need to ask employees to enter their vacation plans, a professor may wish to collect feedback on a particu-

Telegram: @uni_k

www.konkur.in

lar topic from students, or a student organizing an event may wish to allow other students to register for the event, or someone may wish to conduct an online vote on some topic. Google Forms can be used for such activities; your task is to create something like Google Forms, but with authorization on who can fill a form.

Specifically, create a system that will allow users to easily create information collection events. When creating an event, the event creator must define who is eligible to participate; to do so, your system must maintain user information and allow predicates defining a subset of users. The event creator should be able to specify a set of inputs (with types, default values, and validation checks) that the users will have to provide. The event should have an associated deadline, and the system should have the ability to send reminders to users who have not yet submitted their information. The event creator may be given the option of automatic enforcement of the deadline based on a specified date/time, or choosing to login and declare the deadline is over. Statistics about the submissions should be generated—to do so, the event creator may be allowed to create simple summaries on the entered information. The event creator may choose to make some of the summaries public, viewable by all users, either continually (e.g., how many people have responded) or after the deadline (e.g., what was the average feedback score).

Project 9.15 Create a library of functions to simplify creation of web interfaces, using jQuery. You must implement at least the following functions: a function to display a JDBC result set (with tabular formatting), functions to create different types of text and numeric inputs (with validation criteria such as input type and optional range, enforced at the client by appropriate JavaScript code), and functions to create menu items based on a result set. Also implement functions to get input for specified fields of specified relations, ensuring that database constraints such as type and foreign-key constraints are enforced at the client side. Foreign key constraints can also be used to provide either autocomplete or drop-down menus, to ease the task of data entry for the fields.

For extra credit, use support CSS styles which allow the user to change style parameters such as colors and fonts. Build a sample database application to illustrate the use of these functions.

Project 9.16 Design and implement a web-based multiuser calendar system. The system must track appointments for each person, including multioccurrence events, such as weekly meetings, and shared events (where an update made by the event creator gets reflected to all those who share the event). Provide interfaces to schedule multiuser events, where an event creator can add a number of users who are invited to the event. Provide email notification of events. For extra credits implement a web service that can be used by a reminder program running on the client machine.

Telegram: @uni_k

www.konkur.in

Tools

There are several integrated development environments that provide support for web application development. Eclipse (www.eclipse.org) and Netbeans (netbeans.org) are popular open-source IDEs. IntelliJ IDEA (www.jetbrains.com/idea/) is a popular commercial IDE which provides free licenses for students, teachers and non-commercial open source projects. Microsoft's Visual Studio (visualstudio.microsoft.com) also supports web application development. All these IDEs support integration with application servers, to allow web applications to be executed directly from the IDE.

The Apache Tomcat (jakarta.apache.org), Glassfish (javaee.github.io/glassfish/), JBoss Enterprise Application Platform (developers.redhat.com/products/eap/overview/), WildFly (wildfly.org) (which is the community edition of JBoss) and Caucho's Resin (www.caucho.com), are application servers that support servlets and JSP. The Apache web server (apache.org) is the most widely used web server today. Microsoft's IIS (Internet Information Services) is a web and application server that is widely used on Microsoft Windows platforms, supporting Microsoft's ASP.NET (msdn.microsoft.com/asp.net/).

The jQuery JavaScript library jquery.com is among the most widely used JavaScript libraries for creating interactive web interfaces.

Android Studio (developer.android.com/studio/) is a widely used IDE for developing Android apps. XCode (developer.apple.com/xcode/) from Apple and AppCode (www.jetbrains.com/objc/) are popular IDEs for iOS application development. Google's Flutter framework (flutter.io), which is based on the Dart language, and Facebook's React Native (facebook.github.io/react-native/) which is based on Javascript, are frameworks that support cross-platform application development across Android and iOS.

The Open Web Application Security Project (OWASP) (www.owasp.org) provides a variety of resources related to application security, including technical articles, guides, and tools.

Further Reading

The HTML tutorials at www.w3schools.com/html, the CSS tutorials at www.w3schools.com/css are good resources for learning HTML and CSS. A tutorial on Java Servlets can be found at docs.oracle.com/javaee/7/tutorial/servlets.htm. The JavaScript tutorials at www.w3schools.com/js are an excellent source of learning material on JavaScript. You can also learn more about JSON and Ajax as part of the JavaScript tutorial. The jQuery tutorial at www.w3schools.com/Jquery is a very good resource for learning how to use jQuery. These tutorials allow you to modify sample code and test it in the browser, with no software download. Information about the

Telegram: @uni_k

www.konkur.in

.NET framework and about web application development using ASP.NET can be found at msdn.microsoft.com.

You can learn more about the Hibernate ORM and Django (including the Django ORM) from the tutorials and documentation at hibernate.org/orm and docs.djangoproject.com respectively.

The Open Web Application Security Project (OWASP) (www.owasp.org) provides a variety of technical material such as the OWASP Testing Guide, the OWASP Top Ten document which describes critical security risks, and standards for application security verification.

The concepts behind cryptographic hash functions and public-key encryption were introduced in [Diffie and Hellman (1976)] and [Rivest et al. (1978)]. A good reference for cryptography is [Katz and Lindell (2014)], while [Stallings (2017)] provides textbook coverage of cryptography and network security.

Bibliography

[Diffie and Hellman (1976)] W. Diffie and M. E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Volume 22, Number 6 (1976), pages 644–654.

[Katz and Lindell (2014)] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 3rd edition, Chapman and Hall/CRC (2014).

[Rivest et al. (1978)] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, Volume 21, Number 2 (1978), pages 120–126.

[Stallings (2017)] W. Stallings, *Cryptography and Network Security - Principles and Practice*, 7th edition, Pearson (2017).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



PART 4

BIG DATA ANALYTICS

Traditional applications of relational databases are based on structured data and they deal with data from a single enterprise. Modern data management applications often need to deal with data that are not necessarily in relational form; further, such applications also need to deal with volumes of data that are far larger than what a single traditional organization would have generated. In Chapter 10, we study techniques for managing such data, often referred to as Big Data. Our coverage of Big Data in this chapter is from the perspective of a programmer who uses Big Data systems. We start with storage systems for Big Data, and then cover querying techniques, including the MapReduce framework, algebraic operations, steaming data, and graph databases.

One major application of Big Data is data analytics, which refers broadly to the processing of data to infer patterns, correlations, or models for prediction. The financial benefits of making correct decisions can be substantial, as can the costs of making wrong decisions. Organizations therefore make substantial investments both to gather or purchase required data and to build systems for data analytics. In Chapter 11, we cover data analytics in general and, in particular, decision-making tasks that benefit greatly by using data about the past to predict the future and using the predictions to make decisions. Topics covered include data warehousing, online analytical processing, and data mining.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 10



Big Data

Traditional applications of relational databases are based on structured data, and they deal with data from a single enterprise. Modern data management applications often need to deal with data that are not necessarily in relational form; further, such applications also need to deal with volumes of data that are far larger than what a single enterprise would generate. We study techniques for managing such data, often referred to as Big Data, in this chapter.

10.1 Motivation

The growth of the World Wide Web in the 1990s and 2000s resulted in the need to store and query data with volumes that far exceeded the enterprise data that relational databases were designed to manage. Although much of the user-visible data on the web in the early days was static, web sites generated a very large amount of data about users who visited their sites, what web pages they accessed, and when. These data were typically stored on log files on the web server, in textual form. People managing web sites soon realized that there was a wealth of information in the web logs that could be used by companies to understand more about their users and to target advertisements and marketing campaigns at users. Such information included details of which pages had been accessed by users, which could also be linked with user profile data, such as age, gender, income level, and so on, that were collected by many web sites. Transactional web sites such as shopping sites had other kinds of data as well, such as what products a user had browsed or purchased. The 2000s saw exceptionally large growth in the volume of user-generated data, in particular social-media data.

The volume of such data soon grew well beyond the scale that could be handled by traditional database systems, and both storage and processing require a very high degree of parallelism. Furthermore, much of the data were in textual form such as log records, or in other semi-structured forms that we saw in Chapter 8. Such data, are characterized by their size, speed at which they are generated, and the variety of formats, are generically called Big Data.

Telegram: @uni_k

www.konkur.in

Big Data has been contrasted with traditional relational databases on the following metrics:

- **Volume**: The amount of data to be stored and processed is much larger than traditional databases, including traditional parallel relational databases, were designed to handle. Although there is a long history of parallel database systems, early generation parallel databases were designed to work on tens to a few hundreds of machines. In contrast, some of the new applications require the use of thousands of machines in parallel to store and process the data.
- **Velocity**: The rate of arrival of data are much higher in today's networked world than in earlier days. Data management systems must be able to ingest and store data at very high rates. Further, many applications need data items to be processed as they arrive, in order to detect and respond quickly to certain events (such systems are referred to a *streaming data systems*). Thus, processing velocity is very important for many applications today.
- **Variety**: The relational representation of data, relational query languages, and relational database systems have been very successful over the past several decades, and they form the core of the data representation of most organizations. However, clearly, not all data are relational.

  As we saw in Chapter 8, a variety of data representations are used for different purposes today. While much of today's data can be efficiently represented in relational form, there are many data sources that have other forms of data, such as semi-structured data, textual data, and graph data. The SQL query language is well suited to specifying a variety of queries on relational data, and it has been extended to handle semi-structured data. However, many computations cannot be easily expressed in SQL or efficiently evaluated if represented using SQL.

  A new generation of languages and frameworks has been developed for specifying and efficiently executing complex queries on new forms of data.

We shall use the term Big Data in a generic sense, to refer to any data-processing need that requires a high degree of parallelism to handle, regardless of whether the data are relational or otherwise.

Over the past decade, several systems have been developed for storing and processing Big Data, using very large clusters of machines, with thousands, or in some cases, tens of thousands of machines. The term **node** is often used to refer to a machine in a cluster.

10.1.1 Sources and Uses of Big Data

The rapid growth of the web was the key driver for the enormous growth of data volumes in the late 1990s and early 2000s. The initial sources of data were logs from web server software, which recorded user interactions with the web servers. With each user

Telegram: @uni_k

www.konkur.in

clicking on multiple links each day, and hundreds of millions of users, which later grew to billions of users, the large web companies found they were generating multiple terabytes of data each day. Web companies soon realized that there was a lot of important information in the web logs, which could be used for multiple purposes, such as these:

- Deciding what posts, news, and other information to present to which user, to keep them more engaged with the site. Information on what the user had viewed earlier, as well as information on what other users with similar preferences had viewed, are key to making these decisions.
- Deciding what advertisements to show to which users, to maximize the benefit to the advertiser, while also ensuring the advertisements that a user sees are more likely to be of relevance to the user. Again, information on what pages a user had visited, or what advertisements a user had clicked on earlier, are key to making such decisions.
- Deciding how a web site should be structured, to make it easy for most users to find information that they are looking for. Knowing to what pages users typically navigate, and what page they typically view after visiting a particular page, is key to making such decisions.
- Determining user preferences and trends based on page views, which can help a manufacturer or vendor decide what items to produce or stock more of, and what to produce or stock less of. This is part of a more general topic of business intelligence.
- Advertisement display and click-through information. A **click-through** refers to a user clicking on an advertisement to get more information, and is a measure of the success of the advertisement in getting user attention. A **conversion** occurs when the user actually purchases the advertised product or service. Web sites are often paid when a click-through or conversion occurs. This makes click-through and conversion rates for different advertisements a key metric for a site to decide which advertisements to display.

Today, there are many other sources of very high-volume data. Examples include the following:

- Data from mobile phone apps that help in understanding user interaction with the app, in the same way that clicks on a web site help in understanding user interaction with the web site.
- Transaction data from retain enterprises (both online and offline). Early users of very large volumes of data included large retail chains such as Walmart, who used parallel database systems even in the years preceding the web, to manage and analyze their data.

Telegram: @uni_k

www.konkur.in

- Data from sensors. High-end equipment today typically has a large number of sensors to monitor the health of the equipment. Collecting such data centrally helps to track status and predict the chances of problems with the equipment, helping fix problems before they result in failure. The increasing use of such sensors to the connection of sensors and other computing devices embedded within other objects such as vehicles, buildings, machinery, and so forth to the internet, often referred to as the **internet of things**. The number of such devices is now more than the number of humans on the internet.
- Metadata from communication networks, including traffic and other monitoring information for data networks, and call information for voice networks. Such data are important for detecting potential problems before they occur, for detecting problems as they occur, and for capacity planning and other related decisions.

The amount of data stored in databases has been growing rapidly for multiple decades, well before the term Big Data came into use. But the extremely rapid growth of the web created an inflection point, with the major web sites having to handle data generated by hundreds of millions to billions of users; this was a scale significantly greater than most of the earlier applications.

Even companies that are not web related have found it necessary to deal with very large amounts of data. Many companies procure and analyze large volumes of data generated by other companies. For example, web search histories annotated with user profile information, have become available to many companies, which can use such information to make a variety of business decisions, such as planning advertising campaigns, planning what products to manufacture and when, and so on.

Companies today find it essential to make use of social media data to make business decisions. Reactions to new product launches by a company, or a change in existing offerings can be found on Twitter and other social media sites. Not only is the volume of data on social media sites such as Twitter very high, but the data arrives at a very high velocity, and needs to be analyzed and responded to very quickly. For example, if a company puts out an advertisement, and there is strong negative reaction on Twitter, the company would want to detect the issue quickly, and perhaps stop using the advertisement before there is too much damage. Thus, Big Data has become a key enabler for a variety of activities of many organizations today.

10.1.2 Querying Big Data

SQL is by far the most widely used language for querying relational databases. However, there is a wider variety of query language options for Big Data applications, driven by the need to handle more variety of data types, and by the need to scale to very large data volumes/velocity.

Building data management systems that can scale to a large volume/velocity of data requires parallel storage and processing of data. Building a relational database that supports SQL along with other database features, such as transactions (which we

Telegram: @uni_k

www.konkur.in

study later in Chapter 17), and *at the same time* can support very high performance by running on a very large number of machines, is not an easy task. There are two categories of such applications:

1. *Transaction-processing systems that need very high scalability*: Transaction-processing systems support a large number of short running queries and updates.

   It is much easier for a database designed to support transaction processing to scale to very large numbers of machines if the requirements to support all features of a relational database are relaxed. Conversely, many transaction-processing applications that need to scale to very high volumes/velocity can manage without full database support.

   The primary mode of data access for such applications is to store data with an associated key, and to retrieve data with that key; such a storage system is called a key-value store. In the preceding user profile example, the key for user-profile data would be the user's identifier. There are applications that conceptually require joins but implement the joins either in application code or by a form of view materialization.

   For example, in a social-networking application, when a user connects to the system, the user should be shown new posts from all her friends. If the data about posts and friends is maintained in relational format, this would require a join. Suppose that instead, the system maintains an object for each user in a key-value store, containing their friend information as well as their posts. Instead of a join done in the database, the application code could implement the join by first finding the set of friends of the user, and then querying the data object of each friend to find their posts. Another alternative is as follows: whenever a user u_0 makes a post, for each friend u_i of the user, a message is sent to the data object representing u_i, and the data associated with the friend are updated with a summary of the new post. When that user u_i checks for updates, all data required to provide a summary view of posts by friends are available in one place and can be retrieved quickly.

   There are trade-offs between the two alternatives, such as higher cost at query time for the first alternative, versus higher storage cost and higher cost at the time of writes for the second alternative.[1] But both approaches allow the application to carry out its tasks without support for joins in the key-value storage system.

2. *Query processing systems that need very high scalability, and need to support non-relational data*: Typical examples of such systems are those designed to perform analysis on logs generated by web servers and other applications. Other examples include document and knowledge storage and indexing systems, such as those that support keyword search on the web.

[1] It is worth mentioning that it appears (based on limited publicly available information as of 2018) that Facebook uses the first alternative for its news feed to avoid the high storage

Telegram: @uni_k

www.konkur.in

The data consumed by many such applications are stored in multiple files. A system designed to support such applications first needs to be able to store a large number of large files. Second, it must be able to support parallel querying of data stored in such files. Since the data are not necessarily relational, a system designed for querying such data must support arbitrary program code, not just relational algebra or SQL queries.

Big Data applications often require processing of very large volumes of text, image, and video data. Traditionally such data were stored in file systems and processed using stand-alone applications. For example, keyword search on textual data, and its successor, keyword search on the web, both depend on preprocessing textual data, followed by query processing using data structures such as indices built during the preprocessing step. It should be clear that the SQL constructs we have seen earlier are not suited for carrying out such tasks, since the input data are not in relational form, and the output too may not be in relational form.

In earlier days, processing of such data was done using stand-alone programs; this is very similar to how organizational data were processed prior to the advent of database management systems. However, with the very rapid growth of data sizes, the limitations of stand-alone programs became clear. Parallel processing is critical given the very large scale of Big Data. Writing programs that can process data in parallel while dealing with failures (which are common with large scale parallelism) is not easy.

In this chapter, we study techniques for querying of Big Data that are widely used today. A key to the success of these techniques is the fact that they allow specification of complex data processing tasks, while enabling easy parallelization of the tasks. These techniques free the programmer from having to deal with issues such as how to perform parallelization, how to deal with failures, how to deal with load imbalances between machines, and many other similar low-level issues.

10.2 Big Data Storage Systems

Applications on Big Data have extremely high scalability requirements. Popular applications have hundreds of millions of users, and many applications have seen their load increase many-fold within a single year, or even within a few months. To handle the data management needs of such applications, data must be stored partitioned across thousands of computing and storage nodes.

A number of systems for Big Data storage have been developed and deployed over the past two decades to address the data management requirements of such applications. These include the following:

- **Distributed File Systems**. These allow files to be stored across a number of machines, while allowing access to files using a traditional file-system interface. Distributed file systems are used to store large files, such as log files. They are also used as a storage layer for systems that support storage of records.

Telegram: @uni_k

www.konkur.in

- **Sharding across multiple databases.** *Sharding* refers to the process of partitioning of records across multiple systems; in other words, the records are divided up among the systems. A typical use case for sharding is to partition records corresponding to different users across a collection of databases. Each database is a traditional centralized database, which may not have any information about the other databases. It is the job of client software to keep track of how records are partitioned, and to send each query to the appropriate database.
- **Key-Value Storage Systems**. These allow records to be stored and retrieved based on a key, and may additionally provide limited query facilities. However, they are not full-fledged database systems; they are sometimes called NoSQL systems, since such storage systems typically do not support the SQL language.
- **Parallel and Distributed Databases**. These provide a traditional database interface but store data across multiple machines, and they perform query processing in parallel across multiple machines.

Parallel and distributed database storage systems, including distributed file systems and key-value stores, are described in detail in Chapter 21. We provide a user-level overview of these Big Data storage systems in this section.

10.2.1 Distributed File Systems

A **distributed file system** stores files across a large collection of machines while giving a single-file-system view to clients. As with any file system, there is a system of file names and directories, which clients can use to identify and access files. Clients do not need to bother about where the files are stored. Such distributed file systems can store very large amounts of data, and support very large numbers of concurrent clients. Such systems are ideal for storing unstructured data, such as web pages, web server logs, images, and so on, that are stored as large files.

A landmark system in this context was the Google File System (GFS), developed in the early 2000s, which saw widespread use within Google. The open-source Hadoop File System (HDFS) is based on the GFS architecture and is now very widely used.

Distributed file systems are designed for efficient storage of large files, whose sizes range from tens of megabytes to hundreds of gigabytes or more.

The data in a distributed file system is stored across a number of machines. Files are broken up into multiple blocks. The blocks of a single file can be partitioned across multiple machines. Further, each file block is replicated across multiple (typically three) machines, so that a machine failure does not result in the file becoming inaccessible.

File systems, whether centralized or distributed, typically support the following:

- A directory system, which allows a hierarchical organization of files into directories and subdirectories.
- A mapping from a file name to the sequence of identifiers of blocks that store the actual data in each file.

Telegram: @uni_k

www.konkur.in

- The ability to store and retrieve data to/from a block with a specified identifier.

In the case of a centralized file system, the block identifiers help locate blocks in a storage device such as a disk. In the case of a distributed file system, in addition to providing a block identifier, the file system must provide the location (machine identifier) where the block is stored; in fact, due to replication, the file system provides a set of machine identifiers along with each block identifier.

Figure 10.1 shows the architecture of the Hadoop File System (HDFS), which is derived from the architecture of the Google File System (GFS). The core of HDFS is

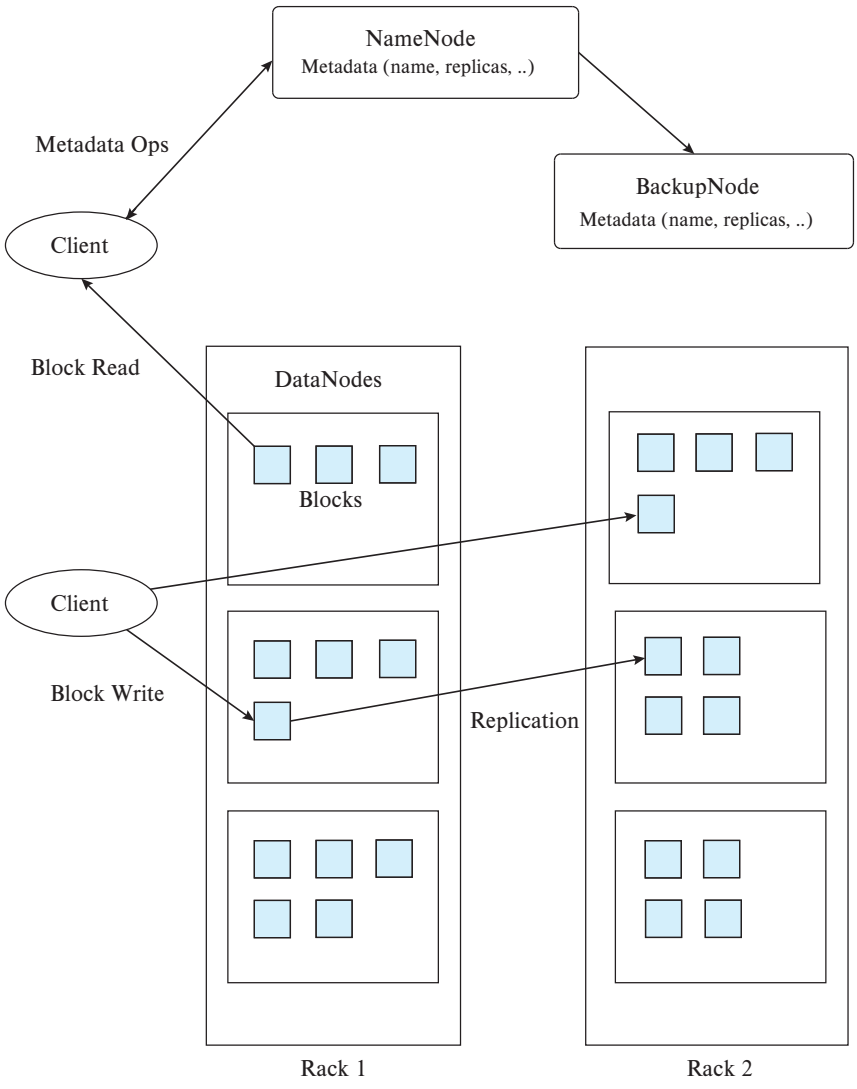


Figure 10.1 Hadoop Distributed File System (HDFS) architecture.

Telegram: @uni_k

www.konkur.in

a server running a machine referred to as the **NameNode**. All file system requests are sent to the NameNode. A file system client program that wants to read an existing file sends the file name (which can be a path, such as /home/avi/book/ch10) to the NameNode. The NameNode stores a list of block identifiers of the blocks in each file; for each block identifier, the NameNode also stores the identifiers of machines that store copies of that block. The machines that store data blocks in HDFS are called **DataNodes**.

For a file read request, the HDFS server sends back a list of block identifiers of the blocks in the file and the identifiers of the machines that contain each block. Each block is then fetched from one of the machines that store a copy of the block.

For a file write, the HDFS server creates new block identifiers and assigns each block identifier to several (usually three) machines, and returns the block identifiers and machine assignment to the client. The client then sends the block identifiers and block data to the assigned machines, which store the data.

Files can be accessed by programs by using HDFS file system APIs that are available in multiple languages, such as Java and Python; the APIs allow a program to connect to the HDFS server and access data.

An HDFS distributed file system can also be connected to the local file system of a machine in such a way that files in HDFS can be accessed as though they are stored locally. This requires providing the address of the NameNode machine, and the port on which the HDFS server listens for requests, to the local file system. The local file system recognizes which file accesses are to files in HDFS based on the file path, and sends appropriate requests to the HDFS server.

More details about distributed file system implementation may be found in Section 21.6.

10.2.2 Sharding

A single database system typically has sufficient storage and performance to handle all the transaction processing needs of an enterprise. However, using a single database is not sufficient for applications with millions or even billions of users, including social-media or similar web-scale applications, but also the user-facing applications of very large organizations such as large banks.

Suppose an organization has built an application with a centralized database, but needs to scale to handle more users, and the centralized database is not capable of handling the storage or processing speed requirements. A commonly used way to deal with such a situation is to partition the data across multiple databases, with a subset of users assigned to each of the databases. The term **sharding** refers to the partitioning of data across multiple databases or machines.

Partitioning is usually done on one or more attributes, referred to as *partitioning attributes*, *partitioning keys*, or *shard keys*. User or account identifiers are commonly used as partitioning keys. Partitioning can be done by defining a range of keys that each of the databases handles; for example, keys from 1 to 100,000 may be assigned to the

Telegram: @uni_k

www.konkur.in

first database, keys from 100,001 to 200,000 to the second database, and so on. Such partitioning is called *range partitioning*. Partitioning may also be done by computing a hash function that maps a key value to a partition number; such partitioning is called *hash partitioning*. We study partitioning of data in detail in Chapter 21.

When sharding is done in application code, the application must keep track of which keys are stored on which database, and must route queries to the appropriate database. Queries that read or update data from multiple databases cannot be processed in a simple manner, since it is not possible to submit a single query that gets executed across all the databases. Instead, the application would need to read data from multiple databases and compute the final query result. Updates across databases cause further issues, which we discuss in Section 10.2.5.

While sharding performed by modifying application code provided a simple way to scale applications, the limitations of the approach soon became apparent. First, the application code has to track how data was partitioned and route queries appropriately. If a database becomes overloaded, parts of the data in that database have to be offloaded to a new database, or to one of the other existing databases; managing this process is a non-trivial task. As more databases are added, there is a greater chance of failure leading to loss of access to data. Replication is needed to ensure data is accessible despite failures, but managing the replicas, and ensuring they are consistent, poses further challenges. Key-value stores, which we study next, address some of these issues. Challenges related to consistency and availability are discussed later, in Section 10.2.5.

10.2.3 Key-Value Storage Systems

Many web applications need to store very large numbers (many billions or in extreme cases, trillions) of relatively small records (of size ranging from a few kilobytes to a few megabytes). Storing each record as a separate file is infeasible, since file systems, including distributed file systems, are not designed to store such large numbers of files.

Ideally, a massively parallel relational database should be used to store such data. However, it is not easy to build relational database systems that can run in parallel across a large number of machines while also supporting standard database features such as foreign-key constraints and transactions.

A number of storage systems have been developed that can scale to the needs of web applications and store large amounts of data, scaling to thousands to tens of thousands of machines, but typically offering only a simple key-value storage interface. A **key-value storage system** (or **key-value store**) is a system that provides a way to store or update a record (value) with an associated key and to retrieve the record with a given key.

Parallel key-value stores partition keys across multiple machines, and route updates and lookups to the correct machine. They also support replication, and ensure that replicas are kept consistent. Further, they provide the ability to add more machines to a system when required, and ensure that the load is automatically balanced across the machines in a system In contrast to systems that implement sharding in the application

Telegram: @uni_k

www.konkur.in

code, systems that use a parallel key-value store do not need to worry about any of the above issues. Parallel key-value stores are therefore more widely used than sharding today.

Widely used parallel key-value stores include Bigtable from Google, Apache HBase, Dynamo from Amazon, Cassandra from Facebook, MongoDB, Azure cloud storage from Microsoft, and Sherpa/PNUTS from Yahoo!, among many others.

While several key-value data stores view the values stored in the data store as an uninterpreted sequence of bytes, and do not look at their content, other data stores allow some form of structure or schema to be associated with each record. Several such key-value storage systems require the stored data to follow a specified data representation, allowing the data store to interpret the stored values and execute simple queries based on stored values. Such data stores are called **document stores**. MongoDB is a widely used data store that accepts values in the JSON format.

Key-value storage systems are, at their core, based on two primitive functions, `put(key, value)`, used to store values with an associated key, and `get(key)`, used to retrieve the stored value associated with the specified key. Some systems, such as Bigtable, additionally provide range queries on key values. Document stores additionally support limited forms of querying on the data values.

An important motivation for the use of key-value stores is their ability to handle very large amounts of data as well as queries, by distributing the work across a *cluster* consisting of a large number of machines. Records are partitioned (divided up) among the machines in the cluster, with each machine storing a subset of the records and processing lookups and updates on those records.

Note that key-value stores are not full-fledged databases, since they do not provide many of the features that are viewed as standard on database systems today. Key-value stores typically do not support declarative querying (using SQL or any other declarative query language) and do not support transactions (which, as we shall see in Chapter 17, allow multiple updates to be committed atomically to ensure that the database state remains consistent despite failures, and control concurrent access to data to ensure that problems do not arise due to concurrent access by multiple transactions). Key-value stores also typically do not support retrieval of records based on selections on non-key attributes, although some document stores do support such retrieval.

An important reason for not supporting such features is that some of them are not easy to support on very large clusters; thus, most systems sacrifice these features in order to achieve scalability. Applications that need scalability may be willing to sacrifice these features in exchange for scalability.

Key-value stores are also called **NoSQL** systems, to emphasize that they do not support SQL, and the lack of support for SQL was initially viewed as something positive, rather than a limitation. However, it soon became clear that lack of database features such as transaction support and support for SQL, make application development more complicated. Thus, many key-value stores have evolved to support features, such as the SQL language and transactions.

Telegram: @uni_k

www.konkur.in


```
show dbs // Shows available databases
use sampledb // Use database sampledb, creating it if it does not exist
db.createCollection("student") // Create a collection
db.createCollection("instructor")
show collections // Shows all collections in the database

db.student.insert({ "id" : "00128", "name" : "Zhang",
      "dept_name" : "Comp. Sci.", "tot_cred" : 102, "advisors" : ["45565"] })
db.student.insert({ "id" : "12345", "name" : "Shankar",
      "dept_name" : "Comp. Sci.", "tot_cred" : 32, "advisors" : ["45565"] })
db.student.insert({ "id" : "19991", "name" : "Brandt",
      "dept_name" : "History", "tot_cred" : 80, "advisors" : [] })
db.instructor.insert({ "id" : "45565", "name" : "Katz",
      "dept_name" : "Comp. Sci.", "salary" : 75000,
      "advisees" : ["00128","12345"] })

db.student.find() // Fetch all students in JSON format
db.student.findOne({"ID": "00128"}) // Find one matching student

db.student.remove({"dept_name": "Comp. Sci."}) // Delete matching students
db.student.drop() // Drops the entire collection
```

Figure 10.2 MongoDB shell commands.

The APIs provided by these systems to store and access data are widely used. While the basic get() and put() functions mentioned earlier are straightforward, most systems support further features. As an example of such APIs, we provide a brief overview of the MongoDB API.

Figure 10.2 illustrates access to the MongoDB document store through a JavaScript shell interface. Such a shell can be opened by executing the mongo command on a system that has MongoDB installed and configured. MongoDB also provides equivalent API functions in a variety of languages, including Java and Python. The use command shown in the figure opens the specified database, creating it if it does not already exist. The db.createCollection() command is used to create collections, which store *documents*; a document in MongoDB is basically a JSON object. The code in the figure creates two collections, student and instructor, and inserts JSON objects representing students and instructors into the two collections.

MongoDB automatically creates identifiers for the inserted objects, which can be used as keys to retrieve the objects. The key associated with an object can be fetched using the _id attribute, and an index on this attribute is created by default.

MongoDB also supports queries based on the stored values. The db.student.find() function returns a collection of all objects in the student collection, while the findOne() function returns one object from the collection. Both functions can take as argument a JSON object that specifies a selection on desired attributes. In our example, the

Telegram: @uni_k

www.konkur.in

student with ID 00128 is retrieved. Similarly, all objects matching such a selection can be deleted by the remove() function shown in the figure. The drop() function shown in the figure drops an entire collection.

MongoDB supports a variety of other features such as creation of indices on specified attributes of the stored JSON objects, such as the ID and name attributes.

Since a key goal of MongoDB is to enable scaling to very large data sizes and query/update loads, MongoDB allows multiple machines to be part of a single MongoDB cluster. Data are then sharded (partitioned) across these machines. We study partitioning of data across machines in detail in Chapter 21, and we study parallel processing of queries in detail in Chapter 22. However we outline key ideas in this section.

In MongoDB (as in many other databases), partitioning is done based on the value of a specified attribute, called the **partitioning attribute** or **shard key**. For example, if we specify that the student collection should be partitioned on the dept_name attribute, all objects of a particular department are stored on one machine, but objects of different departments may be stored on different machines. To ensure data can be accessed even if a machine has failed, each partition is replicated on multiple machines. This way, even if one machine fails, the data in that partition can be fetched from another machine.

Requests from a MongoDB client are sent to a router, which then forwards requests to the appropriate partitions in a cluster.

Bigtable is another key-value store that requires data values to follow a format that allows the storage system access to individual parts of a stored value. In Bigtable, data values (records) can have multiple attributes; the set of attribute names is not predetermined and can vary across different records. Thus, the key for an attribute value conceptually consists of (record-identifier, attribute-name). Each attribute value is just a string as far as Bigtable is concerned. To fetch all attributes of a record, a range query, or more precisely a prefix-match query consisting of just the record identifier, is used. The get() function returns the attribute names along with the values. For efficient retrieval of all attributes of a record, the storage system stores entries sorted by the key, so all attribute values of a particular record are clustered together.

In fact, the record identifier can itself be structured hierarchically, although to Bigtable itself the record identifier is just a string. For example, an application that stores pages retrieved from a web crawl could map a URL of the form:

www.cs.yale.edu/people/silberschatz.html

to the record identifier:

edu.yale.cs.www/people/silberschatz.html

With this representation, all URLs of cs.yale.edu can be retried by a query that fetches all keys with the prefix edu.yale.cs, which would be stored in a consecutive range of key values in the sorted key order. Similarly, all URLs of yale.edu would have a prefix of edu.yale and would be stored in a consecutive range of key values.

Telegram: @uni_k

www.konkur.in

Although Bigtable does not support JSON natively, JSON data can be mapped to the data model of Bigtable. For example, consider the following JSON data:

```
{   "ID": "22222",
    "name": { "firstname: "Albert", "lastname: "Einstein" },
    "deptname": "Physics",
    "children": [
        {"firstname": "Hans", "lastname": "Einstein" },
        {"firstname": "Eduard", "lastname": "Einstein" } ]
}
```

The above data can be represented by a Bigtable record with identifier "22222", with multiple attribute names such as "name.firstname", "deptname", "children[1].firstname" or "children[2].lastname".

Further, a single instance of Bigtable can store data for multiple applications, with multiple tables per application, by simply prefixing the application name and table name to the record identifier.

Many data-storage systems allow multiple versions of data items to be stored. Versions are often identified by timestamp, but they may be alternatively identified by an integer value that is incremented whenever a new version of a data item is created. Lookups can specify the required version of a data item or can pick the version with the highest version number. In Bigtable, for example, a key actually consists of three parts: (record-identifier, attribute-name, timestamp). Bigtable can be accessed as a service from Google. The open-source version of Bigtable, HBase, is widely used.

10.2.4 Parallel and Distributed Databases

Parallel databases are databases that run on multiple machines (together referred to as a cluster) and are designed to store data across multiple machines and to process large queries using multiple machines. Parallel databases were initially developed in the 1980s, and thus they predate the modern generation of Big Data systems. From a programmer viewpoint, parallel databases can be used just like databases running on a single machine.

Early generation parallel databases designed for transaction processing supported only a few machines in a cluster, while those designed to process large analytical queries were designed to support tens to hundreds of machines. Data are replicated across multiple machines in a cluster, to ensure that data are not lost, and they continue to be accessible, even if a machine in a cluster fails. Although failures do occur and need to be dealt with, failures during the processing of a query are not common in systems with tens to hundreds of machines. If a query was being processed on a node that failed, the query is simply restarted, using replicas of data that are on other nodes.

If such database systems are run on clusters with thousands of machines, the probability of failure during execution of a query increases significantly for queries that process a large amount of data and consequently run for a long time. Restarting a query in

Telegram: @uni_k

www.konkur.in

the event of a failure is no longer an option, since there is a fairly high probability that a failure will happen yet again while the query is executing. Techniques to avoid complete restart, allowing only computation on the failed machines to be redone, were developed in the context of *map-reduce* systems, which we study in Section 10.3. However, these techniques introduce significant overhead; given the fact that computation spanning thousands to tens of thousands of nodes is needed only by some exceptionally large applications, even today most parallel relational database systems target applications that run on tens to hundreds of machines and just restart queries in the event of failure.

Query processing in such parallel and distributed databases is covered in detail in Chapter 22, while transaction processing in such databases is covered in Chapter 23.

10.2.5 Replication and Consistency

Replication is key to ensuring availability of data, ensuring a data item can be accessed despite failure of some of the machines storing the data item. Any update to a data item must be applied to all replicas of the data item. As long as all the machines containing the replicas are up and connected to each other, applying the update to all replicas is straightforward.

However, since machines do fail, there are two key problems. The first is how to ensure atomic execution of a transaction that updates data at more than one machine: the transaction execution is said to be atomic if despite failures, either all the data items updated by the transaction are successfully updated, or all the data items are reverted back to their original values. The second problem is, how to perform updates on a data item that has been replicated, when some of the replicas of the data item are on a machine that has failed. A key requirement here is *consistency*, that is, all live replicas of a data item have the same value, and each read sees the latest version of the data item. There are several possible solutions, which offer different degrees of resilience to failures. We study solutions to the both these problems in Chapter 23.

We note that the solutions to the second problem typically require that a majority of the replicas are available for reading and update. If we had 3 replicas, this would require not more than 1 fail, but if we had 5 replicas, even if two machines fail we would still have a majority of replicas available. Under these assumptions, writes will not get blocked, and reads will see the latest value for any data item.

While the probability of multiple machines failing is relatively low, network link failures can cause further problems. In particular, a *network partition* is said to occur if two live machines in a network are unable to communicate with each other.

It has been shown that no protocol can ensure *availability*, that is, the ability to read and write data, while also guaranteeing consistency, in the presence of network partitions. Thus, distributed systems need to make tradeoffs: if they want high availability, they need to sacrifice consistency, for example by allowing reads to see old values of data items, or to allow different replicas to have different values. In the latter case, how to bring the replicas to a common value by merging the updates is a task that the

Telegram: @uni_k

www.konkur.in



application has to deal with. Some applications, or some parts of an application, may choose to prioritize availability over consistency. But other applications, or some parts of an application, may choose to prioritize consistency even at the cost of potential non-availability of the system in the event of failures. The above issues are discussed in more detail in Chapter 23.

Telegram: @uni_k

www.konkur.in

10.3 The MapReduce Paradigm

The MapReduce paradigm models a common situation in parallel processing, where some processing, identified by the map() function, is applied to each of a large number of input records, and then some form of aggregation, identified by the reduce() function, is applied to the result of the map() function. The map() function is also permitted to specify grouping keys, such that the aggregation specified in the reduce() function is applied within each group, identified by the grouping key, of the map() output. We examine the MapReduce paradigm, and the map() and reduce() functions in detail, in the rest of this section.

The MapReduce paradigm for parallel processing has a long history, dating back several decades, in the functional programming and parallel processing community (the map and reduce functions were supported in the Lisp language, for example).

10.3.1 Why MapReduce?

As a motivating example for the use of the MapReduce paradigm, we consider the following *word count* application, which takes a large number of files as input, and outputs a count of the number of times each word appears, across all the files. Here, the input would be in the form of a potentially large number of files stored in a directory.

We start by considering the case of a single file. In this case, it is straightforward to write a program that reads in the words in the file and maintains an in-memory data structure that keeps track of all the words encountered so far, along with their counts. The question is, how to extend the above algorithm, which is sequential in nature, to an environment where there are tens of thousands of files, each containing tens to hundreds of megabytes of data. It is infeasible to process such a large volume of data sequentially.

One solution is to extend the above scheme by coding it as a parallel program that would run across many machines with each machine processing a part of the files. The counts computed locally at each machine must then be combined to get the final counts. In this case, the programmer would be responsible for all the "plumbing" required to start up jobs on different machines, coordinate them, and to compute the final answer. In addition, the "plumbing" code must also deal with ensuring completion of the program in spite of machine failures; failures are quite frequent when the number of participating machines is large, such as in the thousands, and the program runs for a long duration.

The "plumbing" code to implement the above requirements is quite complex; it makes sense to write it just once and reuse it for all desired applications.

MapReduce systems provide the programmer a way of specifying the core logic needed for an application, with the details of the earlier-mentioned plumbing handled by the MapReduce system. The programmer needs to provide only map() and reduce() functions, plus optionally functions for reading and writing data. The map() and reduce() functions provided by the programmer are invoked on the data by the MapRe-

Telegram: @uni_k

www.konkur.in

duce system to process data in parallel. The programmer does not need to be aware of the plumbing or its complexity; in fact, she can for the most part ignore the fact that the program is to be executed in parallel on multiple machines.

The MapReduce approach can be used to process large amounts of data for a variety of applications. The above-mentioned word count program is a toy example of a class of text and document processing applications. Consider, for example, search engines which take keywords and return documents containing the keywords. MapReduce can, for example, be used to process documents and create text indices, which are then used to efficiently find documents containing specified keywords.

10.3.2 MapReduce By Example 1: Word Count

Our word count application can be implemented in the MapReduce framework using the following functions, which we defined in pseudocode. Note that our pseudocode is not in any specific programming language; it is intended to introduce concepts. We describe how to write MapReduce code in specific languages in later sections.

1. In the MapReduce paradigm, the `map()` function provided by the programmer is invoked on each input record and emits zero or more output data items, which are then passed on to the `reduce()` function. The first question is, what is a record? MapReduce systems provide defaults, treating each line of each input file as a record; such a default works well for our word count application, but the programmers are allowed to specify their own functions to break up input files into records.

   For the word count application, the `map()` function could break up each record (line) into individual words and output a number of records, each of which is a pair (*word, count*), where count is the number of occurrences of the word in the record. In fact in our simplified implementation, the `map()` function does even less work and outputs each word as it is found, with a count of 1. These counts are added up later by the `reduce()`. Pseudocode for the `map()` function for the word count program is shown in Figure 10.3.

   The function breaks up the record (line) into individual words.[2] As each word is found, the `map()` function emits (outputs) a record (*word*, 1). Thus, if the file contained just the sentence:

   "One a penny, two a penny, hot cross buns."

   the records output by the `map()` function would be

   ("one", 1), ("a", 1), ("penny", 1),("two", 1), ("a", 1), ("penny", 1), ("hot", 1), ("cross", 1), ("buns", 1).

[2]We omit details of how a line is broken up into words. In a real implementation, non-alphabet characters would be removed, and uppercase characters mapped to lowercase, before breaking up the line based on spaces to generate a list of words.

Telegram: @uni_k

www.konkur.in


```
map(String record) {
      For each word in record
            emit(word, 1).
}

reduce(String key, List value_list) {
      String word = key;
      int count = 0;
      For each value in value_list
            count = count + value
      output(word, count)
}
```

Figure 10.3 Pseudocode of map-reduce job for word counting in a set of files.

In general, the map() function outputs a set of (*key, value*) pairs for each input record. The first attribute (key) of the map() output record is referred to as a **reduce key**, since it is used by the reduce step, which we study next.

2. The MapReduce system takes all the (*key, value*) pairs emitted by the map() functions and sorts (or at least, groups them) such that all records with a particular key are gathered together. All records whose keys match are grouped together, and a list of all the associated values is created. The (*key, list*) pairs are then passed to the reduce() function.

   In our word count example, each key is a word, and the associated list is a list of counts generated for different lines of different files. With our example data, the result of this step is the following:

   ("a", [1,1]), ("buns", [1]) ("cross", [1]), ("hot", [1]), ("one", [1]), ("penny", [1,1]), ("two", [1])

   The reduce() function for our example combines the list of word counts by adding the counts, and outputs (*word, total-count*) pairs. For the example input, the records output by the reduce() function would be as follows:

   ("one", 1), ("a", 2), ("penny", 2), ("two", 1), ("hot", 1), ("cross", 1), ("buns", 1).

   Pseudocode for the reduce() function for the word count program is shown in Figure 10.3. The counts generated by the map() function are all 1, so the reduce() function could have just counted the number of values in the list, but adding up the values allows some optimizations that we will see later.

   A key issue here is that with many files, there may be many occurrences of the same word across different files. Reorganizing the outputs of the map() functions

Telegram: @uni_k

www.konkur.in


```
...
2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 18:26:48.00EST /exer-dir/2.pdf
2013/02/22 18:26:54.00EST /exer-dir/3.pdf
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...
```

Figure 10.4 Log files.

is required to bring all the values for a particular key together. In a parallel system with many machines, this requires data for different reduce keys to be exchanged between machines, so all the values for any particular reduce key are available at a single machine. This work is done by the **shuffle step**, which performs data exchange between machines and then sorts the (*key, value*) pairs to bring all the values for a key together. Observe in our example that the words have actually been sorted alphabetically. Sorting the output records from the map() is one way for the system to collect all occurrences of a word together; the lists for each word are created from the sorted records.

By default, the output of the reduce() function is sent to one or more files, but MapReduce systems allow programmers to control what happens to the output.

10.3.3 MapReduce by Example 2: Log Processing

As another example of the use of the MapReduce paradigm, which is closer to traditional database query processing, suppose we have a log file recording accesses to a web site, which is structured as shown in Figure 10.4. The goal of our file access count application is to find how many times each of the files in the slide-dir directory was accessed between 2013/01/01 and 2013/01/31. The application illustrates one of a variety of kinds of questions an analyst may ask using data from web log files.

For our log-file processing application, each line of the input file can be treated as a record. The map() function would do the following: it would first break up the input record into individual fields, namely date, time, and filename. If the date is in the required date range, the map() function would emit a record (*filename, 1*), which indicates that the filename appeared once in that record. Pseudocode for the map() function for this example is shown in Figure 10.5.

The shuffle step brings all the values for a particular *reduce* key (in our case, a file name) together as a list. The reduce() function provided by the programmer, shown in Figure 10.6, is then invoked for each *reduce* key value. The first argument of reduce() is the *reduce* key itself, while the second argument is a list containing the values in the

Telegram: @uni_k

www.konkur.in


```
map(String record) {
    String attribute[3];
    break up record into tokens (based on space character), and
        store the tokens in array attributes
    String date = attribute[0];
    String time = attribute[1];
    String filename = attribute[2];
    if(date between 2013/01/01 and 2013/01/31
            and filename starts with "http://db-book.com/slide-dir")
        emit(filename, 1).
}
```

Figure 10.5 Pseudocode of map functions for counting file accesses.

records emitted by the map() function for that *reduce* key. In our example, the values for a particular key are added to get the total number of accesses for a file. This number is then output by the reduce() function.

If we were to use the values generated by the map() function, the values would be "1" for all emitted records, and we could have just counted the number of elements in the list. However, MapReduce systems support optimizations such as performing a partial addition of values from each input file, before they are redistributed. In that case, the values received by the reduce() function may not necessarily be ones, and we therefore add the values.

Figure 10.7 shows a schematic view of the flow of keys and values through the map() and reduce() functions. In the figure the mk_i's denote *map* keys, mv_i's denote map input values, rk_i's denote *reduce* keys, and rv_i's denote reduce input values. Reduce outputs are not shown.

```
reduce(String key, List value_list) {
    String filename = key;
    int count = 0;
    For each value in value_list
        count = count + value
    output(filename, count)
}
```

Figure 10.6 Pseudocode of reduce functions for counting file accesses.

Telegram: @uni_k

www.konkur.in

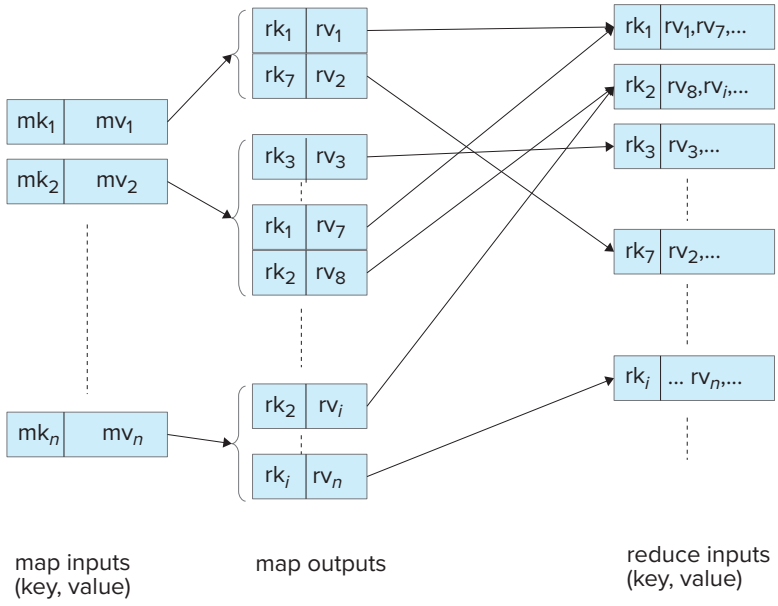


Figure 10.7 Flow of keys and values in a MapReduce job.

10.3.4 Parallel Processing of MapReduce Tasks

Our description of the map() and reduce() functions so far has ignored the issue of parallel processing. We can understand the meaning of MapReduce code without considering parallel processing. However, our goal in using the MapReduce paradigm is to enable parallel processing. Thus, MapReduce systems execute the map() function in parallel on multiple machines, with each map task processing some part of the data, for example some of the files, or even parts of a file in case the input files are very large. Similarly, the reduce() functions are also executed in parallel on multiple machines, with each reduce task processing a subset of the reduce keys (note that a particular call to the reduce() function is still for a single reduce key).

Parallel execution of map and reduce tasks is shown pictorially in Figure 10.8. In the figure, the input file partitions, denoted as Part *i*, could be files or parts of files. The nodes denoted as Map *i* are the map tasks, and the nodes denoted Reduce *i* are the reduce tasks. The master node sends copies of the map() and reduce() code to the map and reduce tasks. The map tasks execute the code and write output data to local files on the machines where the tasks are executed, after being sorted and partitioned based on the reduce key values; separate files are created for each reduce task at each Map node. These files are fetched across the network by the reduce tasks; the files fetched by a reduce task (from different map tasks) are merged and sorted to ensure that all occurrences of a particular reduce key are together in the sorted file. The reduce keys and values are then fed to the reduce() functions.

Telegram: @uni_k

www.konkur.in

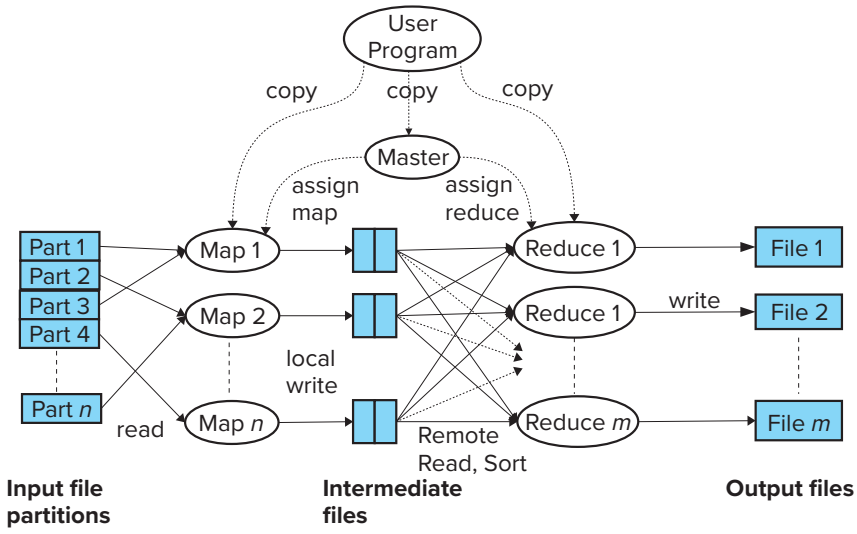


Figure 10.8 Parallel processing of MapReduce job.

MapReduce systems also need to parallelize file input and output across multiple machines; otherwise the single machine storing the data will become a bottleneck. Parallelization of file input and output can be done by using a distributed file system, such as the *Hadoop File System* (*HDFS*). As we saw in Section 10.2, distributed file systems allow a number of machines to cooperate in storing files, partitioning the files across the machines. Further, file system data are replicated (copied) across several (typically three) machines, so that even if a few of the machines fail, the data are available from other machines which have copies of the data in the failed machine.

Today, in addition to distributed file systems such as HDFS, MapReduce systems support input from a variety of Big Data storage systems such as HBase, MongoDB, Cassandra, and Amazon Dynamo, by using storage adapters. Output can similarly be sent to any of these storage systems.

10.3.5 MapReduce in Hadoop

The Hadoop project provides a widely used open-source implementation of MapReduce in the Java language. We summarize its main features here using the Java API provided by Hadoop. We note that Hadoop provides MapReduce APIs in several other languages, such as Python and C++.

Unlike our MapReduce pseudocode, real implementations such as Hadoop require types to be specified for the input keys and values, as well as the output keys and value, of the `map()` function. Similarly, the types of the input as well as output keys and values of the `reduce()` function need to be specified. Hadoop requires the programmer to implement `map()` and `reduce()` functions as member functions of classes that extend Hadoop `Mapper` and `Reducer` classes. Hadoop allows the programmer to provide

Telegram: @uni_k

www.konkur.in

functions to break up the file into records, or to specify that the file is one of the file types for which Hadoop provides built-in functions to break up files into records. For example, the TextInputFormat specifies that the file should be broken up into lines, with each line being a separate record. Compressed file formats are widely used today, with *Avro*, *ORC,* and *Parquet* being the most widely used compressed file formats in the Hadoop world (compressed file formats are discussed in Section 13.6). Decompression is done by the system, and a programmer writing a query need only specify one of the supported types, and the uncompressed representation is made available to the code implementing the query.

Input files in Hadoop can come from a file system of a single machine, but for large datasets, a file system on a single machine would become a performance bottleneck. Hadoop MapReduce allows input and output files to be stored in a distributed file system such as HDFS, allowing multiple machines to read and write data in parallel.

In addition to the reduce() function, Hadoop also allows the programmer to define a combine() function, which can perform a part of the reduce() operation at the node where the map() function is executed. In our word count example, the combine() function would be the same as the reduce() function we saw earlier. The reduce() function would then receive a list of partial counts for a particular word; since the reduce() function for word count adds up the values, it would work correctly even with the combine() function. One benefit of using the combine() function is that it reduces the amount of data that has to be sent over the network: each node that runs map tasks would send only one entry for a word across the network, instead of multiple entries.

A single MapReduce step in Hadoop executes a map and a reduce function. A program may have multiple MapReduce steps, with each step having its own map and reduce functions. The Hadoop API allows a program to execute multiple such MapReduce steps. The reduce() output from each step is written to the (distributed) file system and read back in the following step. Hadoop also allows the programmer to control the number of map and reduce tasks to be run in parallel for the job.

The rest of this section assumes a basic knowledge of Java (you may skip the rest of this section without loss of continuity, if you are not familiar with Java).

Figure 10.9 shows the Java implementation in Hadoop of the word count application we saw earlier. For brevity we have omitted Java import statements. The code defines two classes, one that implements the Mapper interface, and another that implements the Reducer interface. The Mapper and Reducer classes are generic classes which take as arguments the types of the keys and values. Specifically, the generic Mapper and Reducer interfaces both takes four type arguments that specify the types of the input key, input value, output key, and output value, respectively.

The type definition of the Map class in Figure 10.9, which implements the Mapper interface, specifies that the map key is of type LongWritable, is basically a long integer, and the value which is (all or part of) a document is of type Text. The output of map has a key of type Text, since the key is a word, while the value is of type IntWritable, which is an integer value.

Telegram: @uni_k

www.konkur.in


```
public class WordCount {
    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context)
                                    throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }
    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                                    throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
    }
}
```

Figure 10.9 The word count program written in Hadoop.

The map() code for the word count example breaks up the input text value into words using StringTokenizer, and then for each word, it invokes context.write(word,

Telegram: @uni_k

www.konkur.in

one) to output a key and value pair; note that one is an IntWritable object with numeric value 1.

All the values output by the map() invocations that have a particular key (word, in our example) are collected in a list by the MapReduce system infrastructure. Doing so requires interchange of data from multiple map tasks to multiple reduce tasks; in a distributed setting, the data would have to be sent over the network. To ensure that all values for a particular key come together, the MapReduce system typically sorts the keys output by the map functions, ensuring all values for a particular key will come together in the sorted order. This list of values for each key is provided to the reduce() function.

The type of the reduce() input key is the same as the type of the map output key. The reduce() input value in our example is a Java Iterable<IntWritable> object, which contains a list of map output values (IntWritable is the type of the map output value). The output key for reduce() is a word, of type Text, while the output value is a word count, of type IntWritable.

In our example, the reduce() simply adds up the values it receives in its input to get the total count; reduce() writes the word and the total count using the context.write() function.

Note that in our simple example, the values are all 1, so reduce() just needs to count the number of values it receives. In general, however, Hadoop allows the programmer to declare a Combiner class, whose combine() function is run on the output of a single map job; the output of this function replaces multiple map() output values for a single key with a single value. In our example, a combine() function could just count the number of occurrences of each word and output a single value, which is the local word count at the map task. These outputs are then passed on to the reduce() function, which would add up the local counts to get the overall count. The Combiner's job is to reduce the traffic over the network.

A MapReduce job runs a map and a reduce step. A program may have multiple MapReduce steps, and each step would have its own settings for the map and reduce functions. The main() function sets up the parameters for each MapReduce job, and then executes it.

The example code in Figure 10.9 executes a single MapReduce job; the parameters for the job are as follows:

- The classes that contain the map and reduce functions for the job, set by the methods setMapperClass and setReducerClass.
- The types of the job's output key and values, set to Text (for the words) and IntWritable (for the count), respectively, by methods setOutputKeyClass and setOutputValueClass, respectively.
- The input format of the job, set to TextInputFormat by the method job.setInputFormatClass. The default input format in Hadoop is the TextInputFormat, which creates a map key whose value is a byte offset into the file, and the

Telegram: @uni_k

www.konkur.in

map value is the contents of one line of the file. Since files are allowed to be bigger than 4 gigabytes, the offset is of type LongWritable. Programmers can provide their own implementations for the input format class, which would process input files and break the files into records.

- The output format of the job, set to TextOutputFormat by the method job.setOutputFormatClass.
- The directories where the input files are stored, and where the output files must be created, set by the methods addInputPath and addOutputPath.

Hadoop supports many more parameters for MapReduce jobs, such as the number of map and reduce tasks to be run in parallel for the job and the amount of memory to be allocated to each map and reduce task, among many others.

10.3.6 SQL on MapReduce

Many of the applications of MapReduce are for parallel processing of large amounts of non-relational data, using computations that cannot be expressed easily in SQL. For example, our word count program cannot be expressed easily in SQL. There are many real-world uses of MapReduce that cannot be expressed in SQL. Examples include computation of "inverted indices" which are key for web search engines to efficiently answer keyword queries, and computation of Google's PageRank, which is an important measure of the importance of web sites, and is used to rank answers to web search queries.

However, there are a large number of applications that have used the MapReduce paradigm for data processing of various kinds, whose logic can be easily expressed using SQL. If the data were in a database, it would make sense to write such queries using SQL and execute the queries on a parallel database system (parallel database systems are discussed in detail in Chapter 22. Using SQL is much easier for users than is coding in the MapReduce paradigm. However, the data for many such applications reside in a file system, and there are significant time and space overhead demands when loading them into a database.

Relational operations can be implemented using map and reduce steps, as illustrated by the following examples:

- The relational selection operation can be implemented by a single map() function, without a reduce() function (or with a reduce() function that simply outputs its inputs, without any change).
- The relational group by and aggregate function γ can be implemented using a single MapReduce step: the map() outputs records with the group by attribute values as the reduce key; the reduce() function receives a list of all the attribute values for a particular group by key and computes the required aggregate on the values in its input list.

Telegram: @uni_k

www.konkur.in

- A join operation can be implemented using a single MapReduce step, Consider the equijoin operation $r \bowtie_{r.A=s.A} s$. We define a map() function which for each input record r_i outputs a pair $(r_i.A, r_i)$, and similarly for each input record s_i outputs a pair $(s_i.A, s_i)$; the map output also includes a tag to indicate which relation (r or s) the output came from. The reduce() function is invoked for each join-attribute value, with a list of all the r_i and s_i records with that join-attribute value. The function separates out the r and s tuples, and then outputs a cross products of the r tuples and the s tuples, since all of them have the same value for the join attribute.

We leave details as an exercise to the reader (Exercise 10.4). More complex tasks, for example a query with multiple operations, can be expressed using multiple stages of map and reduce tasks.

While it is indeed possible for relational queries to be expressed using the MapReduce paradigm, it can be very cumbersome for a human to do so. Writing queries in SQL is much more concise and easy to understand, but traditional databases did not allow data access from files, nor did they support parallel processing of such queries.

A new generation of systems have been developed that allows queries written in (variants of) the SQL language to be executed in parallel on data stored in file systems. These systems include *Apache Hive* (which was initially developed at Facebook), *SCOPE*, which developed by Microsoft, both of which use variants of SQL, and *Apache Pig* (which was initially developed at Yahoo!), which uses a declarative language called *Pig Latin*, based on the relational algebra. All these systems allow data to be read directly from the file system but allow the programmer to define functions that convert the input data to a record format.

All these systems generate a program containing a sequence of map and reduce tasks to execute a given query. The programs are compiled and executed on a MapReduce framework such as Hadoop. These systems became very popular, and far more queries are written using these systems today than are written directly using the MapReduce paradigm.

Today, Hive implementations provide an option of compiling SQL code to a tree of algebraic operations that are executed on a parallel environment. Apache Tez and Spark are two widely used platforms that support the execution of a tree (or DAG) of algebraic operations on a parallel environment, which we study next in Section 10.4.

10.4 Beyond MapReduce: Algebraic Operations

Relational algebra forms the foundation of relational query processing, allowing queries to be modeled as trees of operations. This idea is extended to settings with more complex data types by supporting algebraic operators that can work on datasets containing records with complex data types, and returning datasets with records containing similar complex data types.

Telegram: @uni_k

www.konkur.in

10.4.1 Motivation for Algebraic Operations

As we saw in Section 10.3.6, relational operations can be expressed by a sequence of map and reduce steps. Expressing tasks in such as fashion can be quite cumbersome.

For example, if programmers need to compute the join of two inputs, they should be able to express it as a single algebraic operation, instead of having to express it indirectly via map and reduce functions. Having access to functions such as joins can greatly simplify the job of a programmer.

The join operation can be executed in parallel, using a variety of techniques that we will see later in Section 22.3. In fact, doing so can be much more efficient than implementing the join using map and reduce functions. Thus, even systems like Hive, where programmers do not directly write MapReduce code, can benefit from direct support for operations such as join.

Later-generation parallel data-processing systems therefore added support for other relational operations such as joins (including variants such as outerjoins and semi-joins), as well as a variety of other operations to support data analytics. For example, many machine-learning models can be modeled as operators that take a set of records as input then output a set of records that have an extra attribute containing the value predicted by the model based on the other attributes of the record. Machine-learning algorithms can themselves be modeled as operators that take a set of training records as input and output a learned model. Processing of data often involves multiple steps, which can be modeled as a sequence (pipeline) or tree of operators.

A unifying framework for these operations is to treat them as *algebraic operations* that take one or more datasets as inputs and output one or more datasets.

Recall that in the relational algebra (Section 2.6) each operation takes one or more relations as input, and outputs a relation. These later-generation parallel query-processing systems are based on the same idea, but there are several differences. A key difference is that the input data could be of arbitrary types, instead of just consisting of columns with atomic data types as in the relational model. Recall that the extended relational algebra required to support SQL could restrict itself to simple arithmetic, string, and boolean expressions. In contrast, the new-generation algebraic operators need to support more complex expressions, requiring the full power of a programming language.

There are a number of frameworks that support algebraic operations on complex data; the most widely used ones today are Apache Tez and Apache Spark.

Apache Tez provides a low-level API which is suitable for system implementors. For example, Hive on Tez compiles SQL queries into algebraic operations that run on Tez. Tez programmers can create trees (or in general Directed Acyclic Graphs, or DAGs) of nodes, and they provide code that is to be executed on each of the nodes. Input nodes would read in data from data sources and pass them to other nodes, which operate on the data. Data can be partitioned across multiple machines, and the code for each node can be executed on each of the machines. Since Tez is not really designed for application programmers to use directly, we do not describe it in further detail.

Telegram: @uni_k

www.konkur.in

However, Apache Spark provides higher-level APIs which are suitable for application programmers. We describe Spark in more detail next.

10.4.2 Algebraic Operations in Spark

Apache Spark is a widely used parallel data processing system that supports a variety of algebraic operations. Data can be input from or output to a variety of storage systems.

Just as relational databases use a relation as the primary abstraction for data representation, Spark uses a representation called a **Resilient Distributed Dataset** (**RDD**), which is a collection of records that can be stored across multiple machines. The term *distributed* refers to the records being stored on different machines, and *resilient* refers to the resilience to failure, in that even if one of the machines fails, records can be retrieved from other machines where they are stored.

Operators in Spark take one or more RDDs as input, and their output is an RDD. The types of records stored in RDDs is not predefined and can be anything that the application desires. Spark also supports a relational data representation called a DataSet, which we describe later.

Spark provides APIs for Java, Scala, and Python. Our coverage of Spark is based on the Java API.

Figure 10.10 shows our word count application, written in Java using Apache Spark; this program uses the RDD data representation, whose Java type is called JavaRDD. Note that JavaRDDs require a type for the record, specified in angular brackets ("<>"). In the program we have RDDs of Java Strings. The program also has JavaPairRDD types, which store records with two attributes of specified types. Records with multiple attributes can be represented by using structured data types instead of primitive data types. While any user-defined data type can be used, the predefined data types Tuple2 which stores two attributes, Tuple3, which stores three attributes, and Tuple4, which stores four attributes, are widely used.

The first step in processing data using Spark is to convert data from input representation to the RDD representation, which is done by the spark.read().textfile() function, which creates a record for each line in the input. Note that the input can be a file or a directory with multiple files; a Spark system running on multiple nodes will actually partition the RDD across multiple machines, although the program can treat it (for most purposes) as if it is a data structure on a single machine. In our sample code in Figure 10.10, the result is the RDD called lines.

The next step in our Spark program is to split each line into an array of words, by calling s.split(" ")) on the line; this function breaks up the line based on spaces and returns an array of words; a more complete function would split the input on other punctuation characters such as periods, semicolons, and so on. The split function can be invoked on each line in the input RDD by calling the map() function, which in Spark returns a single record for each input record. In our example, we instead use a variant called flatMap(), which works as follows: like map(), flatMap() invokes a user-defined

Telegram: @uni_k

www.konkur.in


```
import java.util.Arrays;
import java.util.List;
import scala.Tuple2;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.sql.SparkSession;
public class WordCount {

    public static void main(String[] args) throws Exception {

        if (args.length < 1) {
            System.err.println("Usage: WordCount <file-or-directory-name>");
            System.exit(1);
        }
        SparkSession spark =
            SparkSession.builder().appName("WordCount").getOrCreate();

        JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
        JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator());
        JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));
        JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);

        counts.saveAsTextFile("outputDir"); // Save output files in this directory

        List<Tuple2<String, Integer>> output = counts.collect();
        for (Tuple2<String,Integer> tuple : output) {
            System.out.println(tuple);
        }
        spark.stop();
    }
}
```

Figure 10.10 Word count program in Spark.

function on each input record; the function is expected to return an iterator. A Java iterator supports a `next()` function that can be used to fetch multiple results by calling the function multiple times. The `flatMap()` function invokes the user-defined function to get an iterator, invokes the `next()` function repeatedly on the iterator to get multiple values, and then returns an RDD containing the union of all the values across all input records.

The code shown in Figure 10.10 uses the "lambda expression" syntax introduced in Java 8, which allows functions to be defined compactly, without even giving them a name; in the Java code, the syntax

```
s -> Arrays.asList(s.split(" ")).iterator()
```

Telegram: @uni_k

www.konkur.in

defines a function that takes a parameter s and returns an expression that does the following: it applies the split function described earlier to create an array of words, then uses Arrays.asList to convert the array to a list, and finally applies the iterator() method on the list to create an iterator. The flatMap() function works on this iterator as described earlier.

The result of the above steps is an RDD called words, where each record contains a single word.

The next step is to create a JavaPairRDD called ones, which contains pairs of the form "(word, 1)" for each word in words; if a word appears multiple times in the input file, there would correspondingly be as many records in words and in ones.

Finally the algebraic operation reduceByKey() implements a group by and aggregation step. In the sample code, we specify that addition is to be used for aggregation, by passing the lambda function (i1, i2) -> i1+i2 to the reduceByKey() function. The reduceByKey() function works on a JavaPairRDD, grouping by the first attribute, and aggregating the values of the second attribute using the provided lambda function. When applied on the ones RDD, grouping would be on the word, which is the first attribute, and the values of the second attribute (all ones, in the ones RDD) would be added up. The result is stored in the JavaPairRDD counts.

In general, any binary function can be used to perform the aggregation, as long as it gives the same result regardless of the order in which it is applied on a collection of values.

Finally, the counts RDD is stored to the file system by saveAsTextFile(). Instead of creating just one file, the function creates multiple files if the RDD itself is partitioned across machines.

Key to understanding how parallel processing is achieved is to understand that

- RDDs may be partitioned and stored on multiple machines, and
- each operation may be executed in parallel on multiple machines, on the RDD partition available at the machine. Operations may first repartition their input, to bring related records to the same machine before executing operations in parallel. For example, reduceByKey() would repartition the input RDD to bring all records belonging to a group together on a single machine; records of different groups may be on different machines.

Another important aspect of Spark is that the algebraic operations are not necessarily evaluated immediately on the function call, although the code seems to imply that this is what happens. Instead, the code shown in the figure actually creates a tree of operations; in our code, the leaf operation textFile() reads data from a file; the next operation flatMap() has the textFile() operation as its child; the mapToPairs() in turn has flatMap() as child, and so on. The operators can be thought of in relational terms as defining views, which are not executed as soon as they are defined but get executed later.

Telegram: @uni_k

www.konkur.in

The entire tree of operations actually get evaluated only when certain operations demand that the tree be evaluated. For example, saveAsTextFile() forces the tree to be evaluated; other such functions include collect(), which evaluates the tree and brings all records to a single machine, where they can subsequently be processed, for example by printing them out.

An important benefit of such *lazy evaluation* of the tree (i.e., the tree is evaluated when required, rather than when it is defined) is that before actual evaluation, it is possible for a query optimizer to rewrite the tree to another tree that computes the same result but may execute faster. Query optimization techniques, which we study in Chapter 16 can be applied to optimize such trees.

While the preceding example created a tree of operations, in general the operations may form a *Directed Acyclic Graph* (*DAG*) structure, if the result of an operation is consumed by more than one other operation. That would result in operations having more than one parent, leading to a DAG structure, whereas operations in a tree can have at most one parent.

While RDDs are well suited for representing certain data types such as textual data, a very large fraction of Big Data applications need to deal with structured data, where each record may have multiple attributes. Spark therefore introduced the DataSet type, which supports records with attributes. The DataSet type works well with widely used Parquet, ORC, and Avro file formats (discussed in more detail later in Section 13.6), which are designed to store records with multiple attributes in a compressed fashion. Spark also supports JDBC connectors that can read relations from a database.

The following code illustrates how data in Parquet format can be read and processed in Spark, where spark is a Spark session that has been opened earlier.

```
Dataset<Row> instructor = spark.read().parquet("...");
Dataset<Row> department = spark.read().parquet("...");
instructor.filter(instructor.col("salary").gt(100000))
      .join(department, instructor.col("dept_name")
      .equalTo(department.col("dept_name")))
      .groupBy(department.col("building"))
      .agg(count(instructor.col("ID")));
```

The DataSet<Row> type above uses the type Row, which allows access to column values by name. The code reads *instructor* and *department* relations from Parquet files (whose names are omitted in the code above); Parquet files store metadata such as column names in addition to the values, which allows Spark to create a schema for the relations. The Spark code then applies a filter (selection) operation on the *instructor* relation, which retains only instructors with salary greater than 100000, then joins the result with the *department* relation on the *dept_name* attribute, performs a group by on the *building* attribute (an attribute of the department relation), and for each group (here, each building), a count of the number of *ID* values is computed.

Telegram: @uni_k

www.konkur.in

The ability to define new algebraic operations and to use them in queries has been found to be very useful for many applications and has led to wide adoption of Spark. The Spark system also supports compilation of Hive SQL queries into Spark operation trees, which are then executed.

Spark also allows classes other than Row to be used with DataSets. Spark requires that for each attribute *Attrk* of the class, methods *getAttrk*() and *setAttrk*() must be defined to allow retrieval and storage of attribute values. Suppose we have created a class *Instructor*, and we have a Parquet file whose attributes match those of the class. Then we can read data from Parquet files as follows:

```
Dataset<Instructor> instructor = spark.read().parquet("...").
    as(Encoders.bean(Instructor.class));
```

In this case Parquet provides the names of attributes in the input file, which are used to map their values to attributes of the Instructor class. Unlike with Row, where the types are not known at compile time the types of attributes of Instructor are known at compile time, and can be represented more compactly than if we used the Row type. Further, the methods of the class Instructor can be used to access attributes; for example, we could use getSalary() instead of using col("salary"), which avoids the runtime cost of mapping attribute names to locations in the underlying records. More information on how to use these constructs can be found on the Spark documentation available online at spark.apache.org.

Our coverage of Spark has focused on database operations, but as mentioned earlier, Spark supports a number of other algebraic operations such as those related to machine learning, which can be invoked on DataSet types.

10.5 Streaming Data

Querying of data can be done in an ad hoc manner—for example, whenever an analyst wants to extract some information from a database. It can also be done in a periodic manner—for example, queries may be executed at the beginning of each day to get a summary of transactions that happened on the previous day.

However, there are many applications where queries need to be executed continuously, on data that arrive in a continuous fashion. The term **streaming data** refers to data that arrive in a continuous fashion. Many application domains need to process incoming data in real time (i.e., as they arrive, with delays, if any, guaranteed to be less than some bound).

10.5.1 Applications of Streaming Data

Here are a few examples of streaming data, and the real-time needs of applications that use the data.

Telegram: @uni_k

www.konkur.in

- **Stock market**: In a stock market, each trade that is made (i.e., a stock is sold by someone and bought by someone else) is represented by a tuple. Trades are sent as a stream to processing systems.

  Stock market traders analyze the stream of trades to look for patterns, and they make buy or sell decisions based on the patterns that they observe. Real-time requirements in such systems used to be of the order of seconds in earlier days, but many of today's systems require delays to be of the order of tens of microseconds (usually to be able to react before others do, to the same patterns).

  Stock market regulators may use the same stream, but for a different purpose: to see if there are patterns of trades that are indicative of illegal activities. In both cases, there is a need for continuous queries that look for patterns; the results of the queries are used to carry out further actions.

- **E-commerce**: In an e-commerce site, each purchase made is represented as a tuple, and the sequence of all purchases forms a stream. Further, even the searches executed by a customer are of value to the e-commerce site, even if no actual purchase is made; thus, the searches executed by users form a stream.

  These streams can be used for multiple purposes. For example, if an advertising campaign is launched, the e-commerce site may wish to monitor in real time how the campaign is affecting searches and purchases. The e-commerce site may also wish to detect any spikes in sales of specific products to which it may need to respond by ordering more of that product. Similarly, the site may wish to monitor users for patterns of activities such as frequently returning items, and block further returns or purchases by the user.

- **Sensors**: Sensors are very widely used in systems such as vehicles, buildings, and factories. These sensors send readings periodically, and thus the readings form a stream. Readings in the stream are used to monitor the health of the system. If some readings are abnormal, actions may need to be taken to raise alarms, and to detect and fix any underlying faults, with minimal delay.

  Depending on the complexity of the system and the required frequency of readings, the stream can be of very high volume. In many cases, the monitoring is done at a central facility in the cloud, which monitors a very large number of systems. Parallel processing is essential in such a system to handle very large volumes of data in the incoming streams.

- **Network data**: Any organization that manages a large computer network needs to monitor activity on the network to detect network problems as well as attacks on the network by malicious software (malware). The underlying data being monitored can be represented as a stream of tuples containing data observed by each monitoring point (such as network switch). The tuples could contain information about individual network packets, such as source and destination addresses, size of the packet, and timestamp of packet generation. However, the rate of creation of tuples in such a stream is extremely high, and they cannot be handled except us-

Telegram: @uni_k

www.konkur.in

ing special-purpose hardware. Instead, data can be aggregated to reduce the rate at which tuples are generated: for example, individual tuples could record data such as source and destination addresses, time interval, and total bytes transmitted in the time interval.

This aggregated stream must then be processed to detect problems. For example, link failures could be detected by observing a sudden drop in tuples traversing a particular link. Excessive traffic from multiple hosts to a single or a few destinations could indicate a denial-of-service attack. Packets sent from one host to many other hosts in the network could indicate malware trying to propagate itself to other hosts in the network. Detection of such patterns must be done in real time so that links can be fixed or action taken to stop the malware attack.

- **Social media**: Social media such as Facebook and Twitter get a continuous stream of messages (such as posts or tweets) from users. Each message in the stream of messages must be routed appropriately, for example by sending it to friends or followers. The messages that can potentially be delivered to a subscriber are then ranked and delivered in rank order, based on user preferences, past interactions, or advertisement charges.

  Social-media streams can be consumed not just by humans, but also by software. For example, companies may keep a lookout for tweets regarding the company and raise an alert if there are many tweets that reflect a negative sentiment about the company or its products. If a company launches an advertisement campaign, it may analyze tweets to see if the campaign had an impact on users.

There are many more examples of the need to process and query streaming data across a variety of domains.

10.5.2 Querying Streaming Data

Data stored in a database are sometimes referred to as **data-at-rest**, in contrast to streaming data. In contrast to stored data, streams are unbounded, that is, conceptually they may never end. Queries that can output results only after seeing all the tuples in a stream would then never be able to output any result. As an example, a query that asks for the number of tuples in a stream can never give a final result.

One way to deal with the unbounded nature of streams is to define **windows** on the streams, where each window contains tuples with a certain timestamp range or a certain number of tuples. Given information about timestamps of incoming tuples (e.g., they are increasing), we can infer when all tuples in a particular window have been seen. Based on the above, some query languages for streaming data require that windows be defined on streams, and queries can refer to one or a few windows of tuples rather than to a stream.

Another option is to output results that are correct at a particular point in the stream, but to output updates as more tuples arrive. For example, a count query can

Telegram: @uni_k

www.konkur.in

output the number of tuples seen at a particular point in time, and as more tuples arrive, the query updates its result based on the new count.

Several approaches have been developed for querying streaming data, based on the two options described above.

1. **Continuous queries**. In this approach the incoming data stream is treated as inserts to a relation, and queries on the relations can be written in SQL, or using relational algebra operations. These queries can be registered as **continuous queries**, that is, queries that are running continuously. The result of the query on initial data are output when the system starts up. Each incoming tuple may result in insertion, update, or deletion of tuples in the result of the continuous query. The output of a continuous query is then a stream of updates to the query result, as the underlying database is updated by the incoming streams.

   This approach has some benefits in applications where users wish to view all database inserts that satisfy some conditions. However, a major drawback of the approach is that consumers of a query result would be flooded with a large number of updates to continuous queries if the input rate is high. In particular, this approach is not desirable for applications that output aggregated values, where users may wish to see final aggregates for some period of time, rather than every intermediate result as each incoming tuple is inserted.

2. **Stream query languages**. A second approach is to define a query language by extending SQL or relational algebra, where streams are treated differently from stored relations.

   Most stream query languages use window operations, which are applied to streams, and create relations corresponding to the contents of a window. For example, a window operation on a stream may create sets of tuples for each hour of the day; each such set is thus a relation. Relational operations can be executed on each such set of tuples, including aggregation, selection, and joins with stored relational data or with windows of other streams, to generate outputs.

   We provide an outline of stream query languages in Section 10.5.2.1. These languages separate streaming data from stored relations at the language level and require window operations to be applied before performing relational operations. Doing so ensures that results can be output after seeing only part of a stream. For example, if a stream guarantees that tuples have increasing timestamps, a window based on time can be deduced to have no more tuples once a tuple with a higher timestamp than the window end has been seen. The aggregation result for the window can be output at this point.

   Some streams do not guarantee that tuples have increasing timestamps. However, such streams would contain **punctuations**, that is, metadata tuples that state that all future tuples will have a timestamp greater than some value. Such punctuations are emitted periodically and can be used by window operators to decide

Telegram: @uni_k

www.konkur.in

when an aggregate result, such as aggregates for an hourly window, is complete and can be output.

3. **Algebraic operators on streams**. A third approach is to allow users to write operators (user-defined functions) that are executed on each incoming tuple. Tuples are routed from inputs to operators; outputs of an operator may be routed to another operator, to a system output, or may be stored in a database. Operators can maintain internal state across the tuples that are processed, allowing them to aggregate incoming data. They may also be permitted to store data persistently in a database, for long-term use.

   This approach has seen widespread adoption in recent years, and we describe it in more detail later.

4. **Pattern matching**. A fourth option is to define a pattern matching language and allow users to write multiple rules, each with a pattern and an action. When the system finds a subsequence of tuples that match a particular pattern, the action corresponding to the pattern is executed. Such systems are called **complex event processing** (**CEP**) systems. Popular complex event processing systems include Oracle Event Processing, Microsoft StreamInsight, and FlinkCEP, which is part of the Apache Flink project,

We discuss stream query languages and algebraic operations in more detail later in this section.

Many stream-processing systems keep data in-memory and do not provide persistence guarantees; their goal is to generate results with minimum delay, to enable fast response based on analysis of streaming data. On the other hand, the incoming data may also need to be stored in a database for later processing. To support both patterns of querying, many applications use a so-called **lambda architecture**, where a copy of the input data is provided to the stream-processing system and another copy is provided to a database for storage and later processing. Such an architecture allows stream-processing systems to be developed rapidly, without worrying about persistence-related issues. However, the streaming system and database system are separate, resulting in these problems:

- Queries may need to be written twice, once for the streaming system and once for the database system, in different languages.
- Streaming queries may not be able to access stored data efficiently.

Systems that support streaming queries along with persistent storage and queries that span streams and stored data avoid these problems.

10.5.2.1 Stream Extensions to SQL

SQL window operations were described in Section 5.5.2, but stream query languages support further window types that are not supported by SQL window functions. For

Telegram: @uni_k

www.konkur.in

example, a window that contains tuples for each hour cannot be specified using SQL window functions; note, however, that aggregates on such windows can be specified in SQL in a more roundabout fashion, first computing an extra attribute that contains just the hour component of a timestamp, and then grouping on the hour attribute. Window functions in streaming query languages simplify specification of such aggregation. Commonly supported window functions include:

- **Tumbling window**: Hourly windows are an example of tumbling windows. Windows do not overlap but are adjacent to each other. Windows are specified by their window size (for example, number of hours, minutes, or seconds).
- **Hopping window**: An hourly window computed every 20 minutes would be an example of a hopping window; the window width is fixed, similar to tumbling windows, but adjacent windows can overlap.
- **Sliding window**: Sliding windows are windows of a specified size (based on time, or number of tuples) around each incoming tuple. These are supported by the SQL standard.
- **Session window**: Session windows model users who perform multiple operations as part of a session. A window is identified by a user and a time-out interval, and contains a sequence of operations such that each operation occurs within the time-out interval from the previous operation. For example, if the time-out is 5 minutes, and a user performs an operation at 10 AM, a second operation at 10:04 AM, and a third operation at 11 AM, then the first two operations are part of one session, while the third is part of a different session. A maximum duration may also be specified, so once that duration expires, the session window is closed even if some operations have been performed within the time-out interval.

The exact syntax for specifying windows varies by implementation. Suppose we have a relation *order*(*orderid, datetime, itemid, amount*). In Azure Stream Analytics, the total order amount for each item for each hour can be specified by the following tumbling window:

```
select item, System.Timestamp as window_end, sum(amount)
from order timestamp by datetime
group by itemid, tumblingwindow(hour, 1)
```

Each output tuple has a timestamp whose value is equal to the timestamp of the end of the window; the timestamp can be accessed using the keyword *System.Timestamp* as shown in the query above.

SQL extensions to support streams differentiate between streams, where tuples have implicit timestamps and are expected to receive a potentially unbounded number of tuples and relations whose content is fixed at any point. For example, customers, suppliers, and items associated with orders would be treated as relations, rather than

Telegram: @uni_k

www.konkur.in

as streams. The results of queries with windowing are treated as relations, rather than as streams.

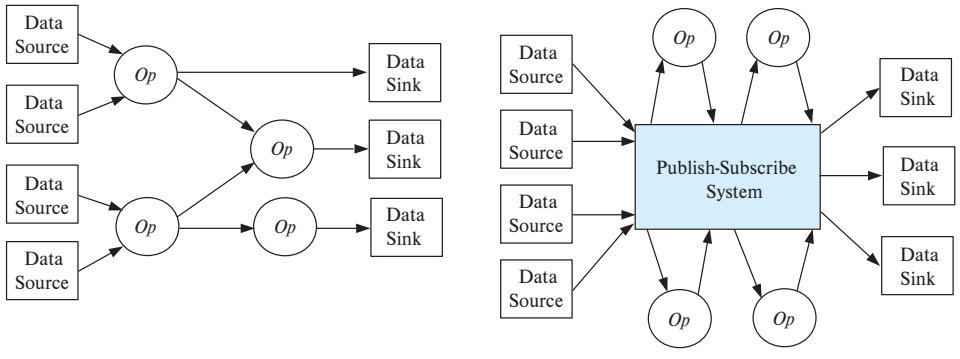
Joins are permitted between a stream and a relation, and the result is a stream; the timestamp of a join result tuple is the same as the timestamp of the input stream tuple. Joins between two streams have the problem that a tuple early in one stream may match a tuple that occurs much later in the other stream; such a join condition would require storing the entire stream for a potentially unbounded amount of time. To avoid this problem, streaming SQL systems allow stream-to-stream join only if there is a join condition that bounds the time difference between matching tuples. A condition that the timestamps of the two tuples differ by at most 1 hour is an example of such a condition.

10.5.3 Algebraic Operations on Streams

While SQL queries on streaming data are quite useful, there are many applications where SQL queries are not a good fit. With the algebraic operations approach to stream processing, user-defined code can be provided for implementing an algebraic operation; a number of predefined algebraic operations, such as selection and windowed aggregation, are also provided.

To perform computation, incoming tuples must be routed to operators that consume the tuples, and outputs of operators must be routed to their consumers. A key task of the implementation is to provide fault-tolerant routing of tuples between system input, operators, and outputs. *Apache Storm* and *Kafka* are widely used implementations that support such routing of data.

The *logical routing* of tuples is done by creating a directed acyclic graph (DAG) with operators as nodes. Edges between nodes define the flow of tuples. Each tuple output by an operator is sent along all the out-edges of the operator, to the consuming operators. Each operator receives tuples from all its in-edges. Figure 10.11a depicts the logical



(a) DAG representation of streaming data flow

(b) Publish-subscribe representation of streaming data flow

Figure 10.11 Routing of streams using DAG and publish-subscribe representations.

Telegram: @uni_k

www.konkur.in

routing of stream tuples through a DAG structure. Operation nodes are denoted as "Op" nodes in the figure. The entry points to the stream-processing system are the data-source nodes of the DAG; these nodes consume tuples from the stream sources and inject them into the stream-processing system. The exit points of the stream-processing system are data-sink nodes; tuples exiting the system through a data sink may be stored in a data store or file system or may be output in some other manner.

One way of implementing a stream-processing system is by specifying the graph as part of the system configuration, which is read when the system starts processing tuples, and is then used to route tuples. The Apache Storm stream-processing system is an example of a system that uses a configuration file to define the graph, which is called a *topology* in the Storm system. Data-source nodes are called *spouts* in the Storm system, while operator nodes are called *bolts*, and edges connect these nodes.

An alternative way of creating such a routing graph is by using **publish-subscribe** systems. A publish-subscribe system allows publication of documents or other forms of data, with an associated topic. Subscribers correspondingly subscribe to specified topics. Whenever a document is published to a particular topic, a copy of the document is sent to all subscribers who have subscribed to that topic. Publish-subscribe systems are also referred to as **pub-sub** systems for short.

When a publish-subscribe system is used for routing tuples in a stream-processing system, tuples are considered documents, and each tuple is tagged with a topic. The entry points to the system conceptually "publish" tuples, each with an associated topic. Operators subscribe to one or more topics; the system routes all tuples with a specific topic to all subscribers of that topic. Operators can also publish their outputs back to the publish-subscribe system, with an associated topic.

A major benefit of the publish-subscribe approach is that operators can be added to the system, or removed from it, with relative ease. Figure 10.11b depicts the routing of tuples using a publish-subscribe representation. Each data source is assigned a unique topic name; the output of each operator is also assigned a unique topic name. Each operator subscribes to the topics of its inputs and publishes to the topics corresponding to its output. Data sources publish to their associated topic, while data sinks subscribe to the topics of the operators whose output goes to the sink.

The Apache Kafka system uses the publish-subscribe model to manage routing of tuples in streams. In the Kafka system, tuples published for a topic are retained for a specified period of time (called the retention period), even if there is currently no subscriber for the topic. Subscribers usually process tuples at the earliest possible time, but in case processing is delayed or temporarily stopped due to failures, the tuples are still available for processing until the retention time expires.

More details of routing, and in particular how publish-subscribe is implemented in a parallel system, are provided in Section 22.8.

The next detail to be addressed is how to implement the algebraic operations. We saw earlier how algebraic operations can be computed using data from files and other data sources as inputs.

Telegram: @uni_k

www.konkur.in

Apache Spark allows streaming data sources to be used as inputs for such operations. The key issue is that some of the operations may not output any results at all until the entire stream is consumed, which may take potentially unbounded time. To avoid this problem, Spark breaks up streams into **discretized streams**, where the stream data for a particular time window are treated as a data input to algebraic operators. When the data in that window have been consumed, the operator generates its output, just as it would have if the data source were a file or a relation.

However, the above approach has the problem that the discretization of streams has to be done before any algebraic operations are executed. Other systems such as Apache Storm and Apache Flink support stream operations, which take a stream as input and output another stream. This is straightforward for operations such as map or relational select operations; each output tuple inherits a timestamp from the input tuple. On the other hand, relational aggregate operations and reduce operations may be unable to generate any output until the entire stream is consumed. To support such operations, Flink supports a window operation which breaks up the stream into windows; aggregates are computed within each window and are output once the window is complete. Note that the output is treated as a stream, where tuples have a timestamp based on the end of the window.[3]

10.6 Graph Databases

Graphs are an important type of data that databases need to deal with. For example, a computer network with multiple routers and links between them can be modeled as a graph, with routers as nodes and network links as edges. Road networks are another common type of graph, with road intersections modeled as nodes and the road links between intersections as edges. Web pages with hyperlinks between them are yet another example of graphs, where web pages can be modeled as nodes and hyperlinks between them as edges.

In fact, if we consider an E-R model of an enterprise, every entity can be modeled as a node of a graph, and every binary relationship can be modeled as an edge of the graph. Ternary and higher-degree relationships are harder to model, but as we saw in Section 6.9.4, such relationships can be modeled as a set of binary relationships if desired.

Graphs can be represented using the relational model using the following two relations:

1. node(ID, label, node_data)
2. edge(fromID, toID, label, edge_data)

where node_data and edge_data contain all the data related to nodes and edges, respectively.

[3]Some systems generate timestamps based on when the window is processed, that are nondeterministic.

Telegram: @uni_k

www.konkur.in

Modeling a graph using just two relations is too simplistic for complex database schemas. For example, applications require modeling of many types of nodes, each with its own set of attributes, and many types of edges, each with its own set of attributes. We can correspondingly have multiple relations that store nodes of different types and multiple relations that store edges of different types.

Although graph data can be easily stored in relational databases, graph databases such as the widely used Neo4j provide several extra features:

- They allow relations to be identified as representing nodes or edges and offer special syntax for defining such relations
- They support query languages designed for easily expressing path queries, which may be harder to express in SQL.
- They provide efficient implementations for such queries, which can execute queries much faster than if they were expressed in SQL and executed on a regular database.
- They provide support for other features such as graph visualization.

As an example of a graph query, we consider a query in the Cypher query language supported by Neo4j. Suppose the input graph has nodes corresponding to students (stored in a relation *student*) and instructors (stored in a relation *instructor*, and an edge type *advisor* from *student* to *instructor*. We omit details of how to create such node and edge types in Neo4j and assume appropriate schemas for these types. We can then write the following query:

```
match (i:instructor)<−[:advisor]−(s:student)
where i.dept_name= 'Comp. Sci.'
return i.ID as ID, i.name as name, collect(s.name) as advisees
```

Observe that the **match** clause in the query connects instructors to students via the advisor relation, which is modeled as a graph path that traverses the advisor edge in the backwards direction (the edge points from student to instructor), by using the syntax (*i*:*instructor*)<−[:*advisor*]−(*s*:*student*). This step basically performs a join of the instructor, advisor and student relations. The query then performs a group by on instructor ID and name, and collects all the students advised by the instructor into a set called advisees. We omit details, and refer the interested reader to online tutorials available at neo4j.com/developer.

Neo4J also supports recursive traversal of edges. For example, suppose we wish to find direct and indirect prerequisites of courses, with the relation *course* modeled with type node, and relation *prereq*(*course_id, prereq_id*) modeled with type edge. We can then write the following query:

```
match (c1:course)−[:prereq *1..]−>(c2:course)
return c1.course_id, c2.course_id
```

Telegram: @uni_k

www.konkur.in

Here, the annotation "*1.." indicates we want to consider paths with multiple *prereq* edges, with a minimum of 1 edge (with a minimum of 0, a course would appear as its own prerequisite).

We note that Neo4j is a centralized system and does not (as of 2018) support parallel processing. However, there are many applications that need to process very large graphs, and parallel processing is key for such applications.

Computation of PageRank (which we saw earlier in Section 8.3.2.2) on very large graphs containing a node for every web page, and an edge for every hyperlink from one page to another, is a good example of a complex computation on very large graphs. The web graph today has hundreds of billions of nodes and trillions of edges. Social networks are another example of very large graphs, containing billions of nodes and edges; computations on such graphs include shortest paths (to find connectivity between people), or computing how influential people are based on edges in the social network.

There are two popular approaches for parallel graph processing:

1. **Map-reduce and algebraic frameworks**: Graphs can be represented as relations, and individual steps of many parallel graph algorithms can be represented as joins. Graphs can thus be stored in a parallel storage system, partitioned across multiple machines. We can then use map-reduce programs, algebraic frameworks such as Spark, or parallel relational database implementations to process each step of a graph algorithm in parallel across multiple nodes.

   Such approaches work well for many applications. However, when performing iterative computations that traverse long paths in graphs, these approaches are quite inefficient, since they typically read the entire graph in each iteration.

2. **Bulk synchronous processing frameworks**: The **bulk synchronous processing** (**BSP**) framework for graph algorithms frames graph algorithms as computations associated with vertices that operate in an iterative manner. Unlike the preceding approach, here the graph is typically stored in memory, with vertices partitioned across multiple machines; most importantly, the graph does not have to be read in each iteration.

   Each vertex (node) of the graph has data (state) associated with it. Similar to how programmers provide `map()` and `reduce()` functions in the MapReduce framework, in the BSP framework programmers provide methods that are executed for each node of the graph. The methods can send messages to neighboring nodes and receive messages from neighboring nodes of the graph. In each iteration, called a **superstep**, the method associated with each node is executed; the method consumes any incoming messages, updates the data associated with the node, and may optionally send messages to neighboring nodes. Messages sent in one iteration are received by the recipients in the next iteration. The method executing at each vertex may vote to halt if they decide they have no more computation to carry out. If in some iteration all vertices vote to halt, and no messages are sent out, the computation can be halted.

Telegram: @uni_k

www.konkur.in

The result of the computation is contained in the state at each node. The state can be collected and output as the result of the computation.

The idea of bulk synchronous processing is quite old but was popularized by the *Pregel* system developed by Google, which provided a fault-tolerant implementation of the framework. The *Apache Giraph* system is an open-source version of the Pregel system.

The GraphX component of Apache Spark supports graph computations on large graphs. It provides an API based on Pregel, as well as a number of other operations that take a graph as input, and output a graph. Operations supported by GraphX include map functions applied on vertices and edges of graphs, join of a graph with an RDD, and an aggregation operation that works as follows: a user-defined function is used to create messages that are sent to all the neighbors of each node, and another user-defined function is used to aggregate the messages. All these operations can be executed in parallel to handle large graphs.

For more information on how to write graph algorithms in such settings, see the references in the Further Reading section at the end of the chapter.

10.7 Summary

- Modern data management applications often need to deal with data that are not necessarily in relational form, and these applications also need to deal with volumes of data that are far larger than what a single traditional organization would have generated.
- The increasing use of data sensors leads to the connection of sensors and other computing devices embedded within other objects to the internet, often referred to as the "internet of things."
- There is now a wider variety of query language options for Big Data applications, driven by the need to handle more varied of data types, and by the need to scale to very large data volumes/velocity.
- Building data management systems that can scale to large volume/velocity of data requires parallel storage and processing of data.
- Distributed file systems allow files to be stored across a number of machines, while allowing access to files using a traditional file-system interface.
- Key-value storage systems allow records to be stored and retrieved based on a key and may additionally provide limited query facilities. These systems are not full-fledged database systems, and they are sometimes called NoSQL systems.
- Parallel and distributed databases provide a traditional database interface, but they store data across multiple machines, and they perform query processing in parallel across multiple machines.

Telegram: @uni_k

www.konkur.in

- The MapReduce paradigm models a common situation in parallel processing, where some processing, identified by the `map()` function, is applied to each of a large number of input records, and then some form of aggregation, identified by the `reduce()` function, is applied to the result of the `map()` function.
- The Hadoop system provides a widely used open-source implementation of MapReduce in the Java language.
- There are a large number of applications that use the MapReduce paradigm for data processing of various kinds whose logic can be easily expressed using SQL.
- Relational algebra forms the foundation of relational query processing, allowing queries to be modeled as trees of operations. This idea is extended to settings with more complex data types by supporting algebraic operators that can work on datasets containing records with complex data types, and returning datasets with records containing similar complex data types.
- There are many applications where queries need to be executed continuously, on data that arrive in a continuous fashion. The term **streaming data** refers to data that arrive in a continuous fashion. Many application domains need to process incoming data in real time.
- Graphs are an important type of data that databases need to deal with.

Review Terms

- Volume
- Velocity
- Conversion
- Internet of things
- Distributed file system
- NameNode server
- DataNodes machines
- Sharding
- Partitioning attribute
- Key-value storage system
- Key-value store
- Document stores
- NoSQL systems
- Shard key
- Parallel databases
- Reduce key
- Shuffle step
- Streaming data
- Data-at-rest
- Windows on the streams
- Continuous queries
- Punctuations
- Lambda architecture
- Tumbling window
- Hopping window
- Sliding window
- Session window
- Publish-subscribe systems
- Pub-sub systems
- Discretized streams
- Superstep

Telegram: @uni_k

www.konkur.in

Practice Exercises

10.1 Suppose you need to store a very large number of small files, each of size say 2 kilobytes. If your choice is between a distributed file system and a distributed key-value store, which would you prefer, and explain why.

10.2 Suppose you need to store data for a very large number of students in a distributed document store such as MongoDB. Suppose also that the data for each student correspond to the data in the *student* and the *takes* relations. How would you represent the above data about students, ensuring that all the data for a particular student can be accessed efficiently? Give an example of the data representation for one student.

10.3 Suppose you wish to store utility bills for a large number of users, where each bill is identified by a customer ID and a date. How would you store the bills in a key-value store that supports range queries, if queries request the bills of a specified customer for a specified date range.

10.4 Give pseudocode for computing a join $r \bowtie_{r.A=s.A} s$ using a single MapReduce step, assuming that the `map()` function is invoked on each tuple of r and s. Assume that the `map()` function can find the name of the relation using `context.relname()`.

10.5 What is the conceptual problem with the following snippet of Apache Spark code meant to work on very large data. Note that the `collect()` function returns a Java collection, and Java collections (from Java 8 onwards) support map and reduce functions.

```
JavaRDD<String< lines = sc.textFile("logDirectory");
int totalLength = lines.collect().map(s -> s.length())
                    .reduce(0,(a,b) -> a+b);
```

10.6 Apache Spark:

a. How does Apache Spark perform computations in parallel?

b. Explain the statement: "Apache Spark performs transformations on RDDs in a lazy manner."

c. What are some of the benefits of lazy evaluation of operations in Apache Spark?

10.7 Given a collection of documents, for each word w_i, let n_i denote the number of times the word occurs in the collection. Let N be the total number of word occurrences across all documents. Next, consider all pairs of consecutive words

Telegram: @uni_k

www.konkur.in

(w_i, w_j) in the document; let $n_{i,j}$ denote the number of occurrences of the word pair (w_i, w_j) across all documents.

Write an Apache Spark program that, given a collection of documents in a directory, computes N, all pairs (w_i, n_i), and all pairs $((w_i, w_j), n_{i,j})$. Then output all word pairs such that $n_{i,j}/N \geq 10 * (n_i/N) * (n_j/N)$. These are word pairs that occur 10 times or more as frequently as they would be expected to occur if the two words occurred independently of each other.

You will find the join operation on RDDs useful for the last step, to bring related counts together. For simplicity, do not bother about word pairs that cross lines. Also assume for simplicity that words only occur in lowercase and that there are no punctuation marks.

10.8 Consider the following query using the tumbling window operator:

select *item*, *System.Timestamp* **as** *window_end*, **sum**(*amount*)
from *order* **timestamp by** *datetime*
group by *itemid*, **tumblingwindow**(*hour*, 1)

Give an equivalent query using normal SQL constructs, without using the tumbling window operator. You can assume that the timestamp can be converted to an integer value that represents the number of seconds elapsed since (say) midnight, January 1, 1970, using the function *to_seconds*(*timestamp*). You can also assume that the usual arithmetic functions are available, along with the function *floor*(a) which returns the largest integer $\leq a$.

10.9 Suppose you wish to model the university schema as a graph. For each of the following relations, explain whether the relation would be modeled as a node or as an edge:

(i) *student*, (ii) *instructor*, (iii) *course*, (iv) *section*, (v) *takes*, (vi) *teaches*.

Does the model capture connections between sections and courses?

Exercises

10.10 Give four ways in which information in web logs pertaining to the web pages visited by a user can be used by the web site.

10.11 One of the characteristics of Big Data is the variety of data. Explain why this characteristic has resulted in the need for languages other than SQL for processing Big Data.

10.12 Suppose your company has built a database application that runs on a centralized database, but even with a high-end computer and appropriate indices created on the data, the system is not able to handle the transaction load, lead-

Telegram: @uni_k

www.konkur.in

ing to slow processing of queries. What would be some of your options to allow the application to handle the transaction load?

10.13 The map-reduce framework is quite useful for creating inverted indices on a set of documents. An inverted index stores for each word a list of all document IDs that it appears in (offsets in the documents are also normally stored, but we shall ignore them in this question).

For example, if the input document IDs and contents are as follows:

1: data clean
2: data base
3: clean base

then the inverted lists would

data: 1, 2
clean: 1, 3
base: 2, 3

Give pseudocode for map and reduce functions to create inverted indices on a given set of files (each file is a document). Assume the document ID is available using a function context.getDocumentID(), and the map function is invoked once per line of the document. The output inverted list for each word should be a list of document IDs separated by commas. The document IDs are normally sorted, but for the purpose of this question you do not need to bother to sort them.

10.14 Fill in the blanks below to complete the following Apache Spark program which computes the number of occurrences of each word in a file. For simplicity we assume that words only occur in lowercase, and there are no punctuation marks.

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts =
    textFile._____(s -> Arrays.asList(s.split(" "))._____())
     .mapToPair(word -> new _____).reduceByKey((a, b) -> a + b);
```

10.15 Suppose a stream can deliver tuples out of order with respect to tuple timestamps. What extra information should the stream provide, so a stream query processing system can decide when all tuples in a window have been seen?

10.16 Explain how multiple operations can be executed on a stream using a publish-subscribe system such as Apache Kafka.

Tools

A wide variety of open-source Big Data tools are available, in addition to some commercial tools. In addition, a number of these tools are available on cloud plat-

Telegram: @uni_k

www.konkur.in

forms. We list below several popular tools, along with the URLs where they can be found. Apache HDFS (hadoop.apache.org) is a widely used distributed file system implementation. Open-source distributed/parallel key-value stores include Apache HBase (hbase.apache.org), Apache Cassandra (cassandra.apache.org), MongoDB (www.mongodb.com), and Riak (basho.com).

Hosted cloud storage systems include the Amazon S3 storage system (aws.amazon.com/s3) and Google Cloud Storage (cloud.google.com/storage). Hosted key-value stores include Google BigTable (cloud.google.com/bigtable), and Amazon DynamoDB (aws.amazon.com/dynamodb).

Google Spanner (cloud.google.com/spanner) and the open source CockroachDB (www.cockroachlabs.com) are scalable parallel databases that support SQL and transactions, and strongly consistent storage.

Open-source MapReduce systems include Apache Hadoop (hadoop.apache.org), and Apache Spark (spark.apache.org), while Apache Tez (tez.apache.org) supports data processing using a DAG of algebraic operators. These are also available as cloud-based offerings from Amazon Elastic MapReduce (aws.amazon.com/emr), which also supports Apache HDFS and Apache HBase, and from Microsoft Azure (azure.microsoft.com).

Apache Hive (hive.apache.org) is a popular open-source SQL implementation that runs on top of the Apache MapReduce, Apache Tez, as well as Apache Spark; these systems are designed to support large queries running in parallel on multiple machines. Apache Impala (impala.apache.org) is an SQL implementation that runs on Hadoop, and is designed to handle a large number of queries, and to return query results with minimal delays (latency). Hosted SQL offerings on the cloud that support parallel processing include Amazon EMR (aws.amazon.com/emr), Google Cloud SQL (cloud.google.com/sql), and Microsoft Azure SQL (azure.microsoft.com).

Apache Kafka (kafka.apache.org) and Apache Flink (flink.apache.org) are open-source stream-processing systems; Apache Spark also provides support for stream processing. Hosted stream-processing platforms include Amazon Kinesis (aws.amazon.com/kinesis), Google Cloud Dataflow (cloud.google.com/dataflow) and Microsoft Stream Analytics (azure.microsoft.com). Open source graph processing platforms include Neo4J (neo4j.com) and Apache Giraph (giraph.apache.org).

Further Reading

[Davoudian et al. (2018)] provides a nice survey of NoSQL data stores, including data model querying and internals. More information about Apache Hadoop, including documentation on HDFS and Hadoop MapReduce, can be found on the Apache Hadoop homepage, hadoop.apache.org. Information about Apache Spark may be found on the Spark homepage, spark.apache.org. Information about the Apache Kafka streaming data platform may be found on kafka.apache.org, and details of stream processing in Apache Flink may be found on flink.apache.org. Bulk Synchronous Processing

Telegram: @uni_k

www.konkur.in

was introduced in [Valiant (1990)]. A description of the Pregel system, including its support for bulk synchronous processing, may be found in [Malewicz et al. (2010)], while information about the open source equivalent, Apache Giraph, may be found on giraph.apache.org.

Bibliography

[Davoudian et al. (2018)] A. Davoudian, L. Chen, and M. Liu, "A Survey of NoSQL Stores", *ACM Computing Surveys*, Volume 51, Number 2 (2018), pages 2–42.

[Malewicz et al. (2010)] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2010), pages 135–146.

[Valiant (1990)] L. G. Valiant, "A Bridging Model for Parallel Computation", *Communications of the ACM*, Volume 33, Number 8 (1990), pages 103–111.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 11



Data Analytics

Decision-making tasks benefit greatly by using data about the past to predict the future and using the predictions to make decisions. For example, online advertisement systems need to decide what advertisement to show to each user. Analysis of past actions and profiles of other users, as well as past actions and profile of the current user, are key to deciding which advertisement the user is most likely to respond to. Here, each decision is low-value, but with high volumes the overall value of making the right decisions is very high. At the other end of the value spectrum, manufacturers and retailers need to decide what items to manufacture or stock many months ahead of the actual sale of the items. Predicting future demand of different types of items based on past sales and other indicators is key to avoiding both overproduction or overstocking of some items, and underproduction or understocking of other items. Errors can lead to monetary loss due to unsold inventory of some items, or loss of potential revenue due to nonavailability of some items.

The term **data analytics** refers broadly to the processing of data to infer patterns, correlations, or models for prediction. The results of analytics are then used to drive business decisions.

The financial benefits of making correct decisions can be substantial, as can the costs of making wrong decisions. Organizations therefore invest a lot of money to gather or purchase required data and build systems for data analytics.

11.1 Overview of Analytics

Large companies have diverse sources of data that they need to use for making business decisions. The sources may store the data under different schemas. For performance reasons (as well as for reasons of organization control), the data sources usually will not permit other parts of the company to retrieve data on demand.

Organizations therefore typically gather data from multiple sources into one location, referred to as a *data warehouse*. Data warehouses gather data from multiple sources at a single site, under a unified schema (which is usually designed to support efficient analysis, even at the cost of redundant storage). Thus, they provide the user

Telegram: @uni_k

www.konkur.in

a single uniform interface to data. However, data warehouses today also collect and store data from non-relational sources, where schema unification is not possible. Some sources of data have errors that can be detected and corrected using business constraints; further, organizations may collect data from multiple sources, and there may be duplicates in the data collected from different sources. These steps of collecting data, cleaning/deduplicating the data, and loading the data into a warehouse are referred to as *extract, transform and load* (ETL) tasks. We study issues in building and maintaining a data warehouse in Section 11.2.

The most basic form of analytics is generation of aggregates and reports summarizing the data in ways that are meaningful to the organization. Analysts need to get a number of different aggregates and compare them to understand patterns in the data. Aggregates, and in some cases the underlying data, are typically presented graphically as charts, to make it easy for humans to visualize the data. Dashboards that display charts summarizing key organizational parameters, such as sales, expenses, product returns, and so forth, are popular means of monitoring the health of an organization. Analysts also need to visualize data in ways that can highlight anomalies or give insights into causes for changes in the business.

Systems that support very efficient analysis, where aggregate queries on large data are answered in almost real time (as opposed to being answered after tens of minutes or multiple hours) are popular with analysts. Such systems are referred to as *online analytical processing* (OLAP) systems. We discuss online analytical processing in Section 11.3, where we cover the concept of multidimensional data, OLAP operations, relational representation of multidimensional summaries. We also discuss graphical representation of data and visualization in Section 11.3.

Statistical analysis is an important part of data analysis. There are several tools that are designed for statistical analysis, including the R language/environment, which is open source, and commercial systems such as SAS and SPSS. The R language is widely used today, and in addition to features for statistical analysis, it supports facilities for graphical display of data. A large number of R packages (libraries) are available that implement a wide variety of data analysis tasks, including many machine-learning algorithms. R has been integrated with databases as well as with Big Data systems such as Apache Spark, which allows R programs to be executed in parallel on large datasets. Statistical analysis is a large area by itself, and we do not discuss it further in this book. References providing more information may be found in the Further Reading section at the end of this chapter.

Prediction of different forms is another key aspect of analytics. For example, banks need to decide whether to give a loan to a loan applicant, and online advertisers need to decide which advertisement to show to a particular user. As another example, manufacturers and retailers need to make decisions on what items to manufacture or order in what quantities.

These decisions are driven significantly by techniques for analyzing past data and using the past to predict the future. For example, the risk of loan default can be predicted as follows. First, the bank would examine the loan default history of past cus-

Telegram: @uni_k

www.konkur.in

tomers, find key features of customers, such as salary, education level, job history, and so on, that help in prediction of loan default. The bank would then build a prediction model (such as a decision tree, which we study later in this chapter) using the chosen features. When a customer then applies for a loan, the features of that particular customer are fed into the model which makes a prediction, such as an estimated likelihood of loan default. The prediction is used to make business decisions, such as whether to give a loan to the customer.

Similarly, analysts may look at the past history of sales and use it to predict future sales, to make decisions on what and how much to manufacture or order, or how to target their advertising. For example, a car company may search for customer attributes that help predict who buys different types of cars. It may find that most of its small sports cars are bought by young women whose annual incomes are above $50,000. The company may then target its marketing to attract more such women to buy its small sports cars and may avoid wasting money trying to attract other categories of people to buy those cars.

Machine-learning techniques are key to finding patterns in data and in making predictions from these patterns. The field of *data mining* combines knowledge-discovery techniques invented by machine-learning researchers with efficient implementation techniques that enable them to be used on extremely large databases. Section 11.4 discusses data mining.

The term **business intelligence** (**BI**) is used in a broadly similar sense to data analytics. The term **decision support** is also used in a related but narrower sense, with a focus on reporting and aggregation, but not including machine learning/data mining. Decision-support tasks typically use SQL queries to process large amounts of data. Decision-support queries are can be contrasted with queries for *online transaction processing*, where each query typically reads only a small amount of data and may perform a few small updates.

11.2 Data Warehousing

Large organizations have a complex internal organization structure, and therefore different data may be present in different locations, or on different operational systems, or under different schemas. For instance, manufacturing-problem data and customer-complaint data may be stored on different database systems. Organizations often purchase data from external sources, such as mailing lists that are used for product promotions, or credit scores of customers that are provided by credit bureaus, to decide on creditworthiness of customers.[1]

Corporate decision makers require access to information from multiple such sources. Setting up queries on individual sources is both cumbersome and inefficient.

[1] Credit bureaus are companies that gather information about consumers from multiple sources and compute a creditworthiness score for each consumer.

Telegram: @uni_k

www.konkur.in

Moreover, the sources of data may store only current data, whereas decision makers may need access to past data as well; for instance, information about how purchase patterns have changed in the past few years could be of great importance. Data warehouses provide a solution to these problems.

A **data warehouse** is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write. Moreover, by accessing information for decision support from a data warehouse, the decision maker ensures that online transaction-processing systems are not affected by the decision-support workload.

11.2.1 Components of a Data Warehouse

Figure 11.1 shows the architecture of a typical data warehouse and illustrates the gathering of data, the storage of data, and the querying and data analysis support. Among the issues to be addressed in building a warehouse are the following:

- **When and how to gather data.** In a **source-driven architecture** for gathering data, the data sources transmit new information, either continually (as transaction processing takes place), or periodically (nightly, for example). In a **destination-driven architecture**, the data warehouse periodically sends requests for new data to the sources.

  Unless updates at the sources are "synchronously" replicated at the warehouse, the warehouse will never be quite up-to-date with the sources. Synchronous replication can be expensive, so many data warehouses do not use synchronous replication, and they perform queries only on data that are old enough that they have

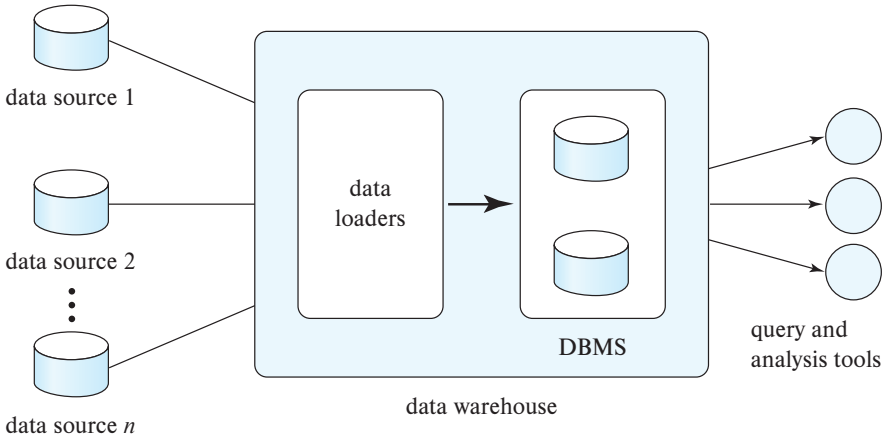


Figure 11.1 Data-warehouse architecture.

Telegram: @uni_k

www.konkur.in

been completely replicated. Traditionally, analysts were happy with using yesterday's data, so data warehouses could be loaded with data up to the end of the previous day. However, increasingly organizations want more up-to-date data. The data freshness requirements depend on the application. Data that are within a few hours old may be sufficient for some applications; others that require real-time responses to events may use stream processing infrastructure (described in Section 10.5) instead of depending on a warehouse infrastructure.

- **What schema to use.** Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration and to convert data to the integrated schema before they are stored. As a result, the data stored in the warehouse are not just a copy of the data at the sources. Instead, they can be thought of as a materialized view of the data at the sources.

- **Data transformation and cleansing.** The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, which can be corrected. For example, names are often misspelled, and addresses may have street, area, or city names misspelled, or postal codes entered incorrectly. These can be corrected to a reasonable extent by consulting a database of street names and postal codes in each city. The approximate matching of data required for this task is referred to as **fuzzy lookup**.

  Address lists collected from multiple sources may have duplicates that need to be eliminated in a **merge–purge operation** (this operation is also referred to as **deduplication**). Records for multiple individuals in a house may be grouped together so only one mailing is sent to each house; this operation is called **householding**.

  Data may be **transformed** in ways other than cleansing, such as changing the units of measure, or converting the data to a different schema by joining data from multiple source relations. Data warehouses typically have graphical tools to support data transformation. Such tools allow transformation to be specified as boxes, and edges can be created between boxes to indicate the flow of data. Conditional boxes can route data to an appropriate next step in transformation.

- **How to propagate updates**. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward. If they are not, the problem of propagating updates is basically the *view-maintenance* problem, which was discussed in Section 4.2.3, and is covered in more detail in Section 16.5.

- **What data to summarize.** The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation. For example, instead of storing data about every sale of clothing, we can store total sales of clothing by item name and category.

Telegram: @uni_k

www.konkur.in

The different steps involved in getting data into a data warehouse are called **extract, transform, and load** or **ETL** tasks; extraction refers to getting data from the sources, while load refers to loading the data into the data warehouse. In current generation data warehouses that support user-defined functions or MapReduce frameworks, data may be extracted, loaded into the warehouse, and then transformed. The steps are then referred to as **extract, load, and transform** or **ELT** tasks. The ELT approach permits the use of parallel processing frameworks for data transformation.

11.2.2 Multidimensional Data and Warehouse Schemas

Data warehouses typically have schemas that are designed for data analysis, using tools such as OLAP tools. The relations in a data **warehouse schema** can usually be classified as *fact tables* and *dimension tables*. **Fact tables** record information about individual events, such as sales, and are usually very large. A table recording sales information for a retail store, with one tuple for each item that is sold, is a typical example of a fact table. The attributes in fact table can be classified as either *dimension attributes* or *measure attributes*, The **measure attributes** store quantitative information, which can be aggregated upon; the measure attributes of a *sales* table would include the number of items sold and the price of the items. In contrast, **dimension attributes** are dimensions upon which measure attributes, and summaries of measure attributes, are grouped and viewed. The dimension attributes of a *sales* table would include an item identifier, the date when the item is sold, which location (store) the item was sold from, the customer who bought the item, and so on.

Data that can be modeled using dimension attributes and measure attributes are called **multidimensional data**.

To minimize storage requirements, dimension attributes are usually short identifiers that are foreign keys into other tables called **dimension tables**. For instance, a fact table *sales* would have dimension attributes *item_id*, *store_id*, *customer_id*, and *date*, and measure attributes *number* and *price*. The attribute *store_id* is a foreign key into a dimension table *store*, which has other attributes such as store location (city, state, country). The *item_id* attribute of the *sales* table would be a foreign key into a dimension table *item_info*, which would contain information such as the name of the item, the category to which the item belongs, and other item details such as color and size. The *customer_id* attribute would be a foreign key into a *customer* table containing attributes such as name and address of the customer. We can also view the *date* attribute as a foreign key into a *date_info* table giving the month, quarter, and year of each date.

The resultant schema appears in Figure 11.2. Such a schema, with a fact table, multiple dimension tables, and foreign keys from the fact table to the dimension tables is called a **star schema**. More complex data-warehouse designs may have multiple levels of dimension tables; for instance, the *item_info* table may have an attribute *manufacturer_id* that is a foreign key into another table giving details of the manufacturer. Such schemas are called **snowflake schemas**. Complex data-warehouse designs may also have more than one fact table.

Telegram: @uni_k

www.konkur.in

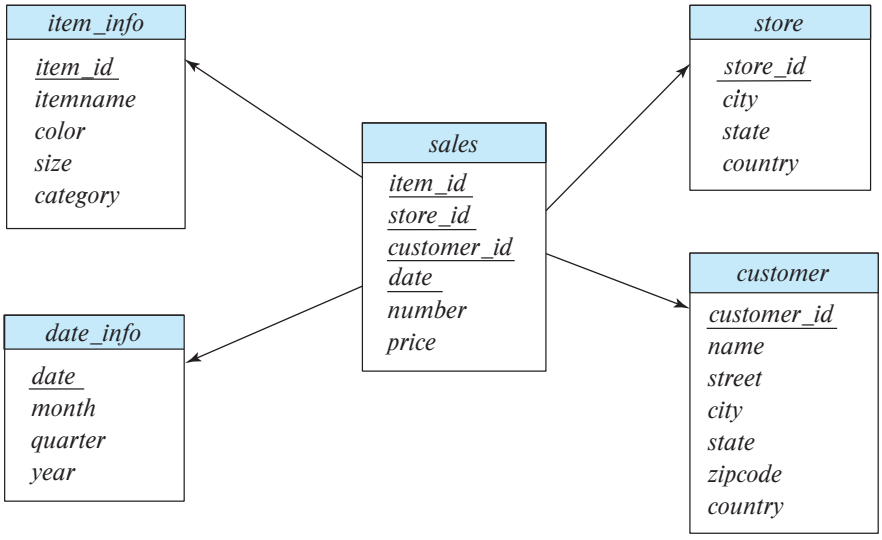


Figure 11.2 Star schema for a data warehouse.

11.2.3 Database Support for Data Warehouses

The requirements of a database system designed for transaction processing are somewhat different from one designed to support a data-warehouse system. One key difference is that a transaction-processing database needs to support many small queries, which may involve updates in addition to reads. In contrast, data warehouses typically need to process far fewer queries, but each query accesses a much larger amount of data.

Most importantly, while new records are inserted into relations in a data warehouse, and old records may be deleted once they are no longer needed, to make space for new data, records are typically never updated once they are added to a relation. Thus, data warehouses do not need to pay any overhead for concurrency control. (As described in Chapter 17 and Chapter 18, if concurrent transactions read and write the same data, the resultant data may become inconsistent. Concurrency control restricts concurrent accesses in a way that ensures there is no erroneous update to the database.) The overhead of concurrency control can be significant in terms of not just time taken for query processing, but also in terms of storage, since databases often store multiple versions of data to avoid conflicts between small update transactions and long read-only transactions. None of these overheads are needed in a data warehouse.

Databases traditionally store all attributes of a tuple together, and tuples are stored sequentially in a file. Such a storage layout is referred to as *row-oriented storage*. In contrast, in **column-oriented storage**, each attribute of a relation is stored in a separate file, with values from successive tuples stored at successive positions in the file. Assuming fixed-size data types, the value of attribute A of the ith tuple of a relation can be found

Telegram: @uni_k

www.konkur.in

by accessing the file corresponding to attribute A and reading the value at offset $(i - 1)$ times the size (in bytes) of values in attribute A.

Column-oriented storage has at least two major benefits over row-oriented storage:

1. When a query needs to access only a few attributes of a relation with a large number of attributes, the remaining attributes need not be fetched from disk into memory. In contrast, in row-oriented storage, not only are irrelevant attributes fetched into memory, but they may also get prefetched into processor cache, wasting cache space and memory bandwidth, if they are stored adjacent to attributes used in the query.
2. Storing values of the same type together increases the effectiveness of compression; compression can greatly reduce both the disk storage cost and the time to retrieve data from disk.

On the other hand, column-oriented storage has the drawback that storing or fetching a single tuple requires multiple I/O operations.

As a result of these trade-offs, column-oriented storage is not widely used for transaction-processing applications. However, column-oriented storage is today widely used for data-warehousing applications, where accesses are rarely to individual tuples but rather require scanning and aggregating multiple tuples. Column-oriented storage is described in more detail in Section 13.6.

Database implementations that are designed purely for data warehouse applications include Teradata, Sybase IQ, and Amazon Redshift. Many traditional databases support efficient execution of data warehousing applications by adding features such as columnar storage; these include Oracle, SAP HANA, Microsoft SQL Server, and IBM DB2.

In the 2010s there has been an explosive growth in Big Data systems that are designed to process queries over data stored in files. Such systems are now a key part of the data warehouse infrastructure. As we saw in Section 10.3, the motivation for such systems was the growth of data generated by online systems in the form of log files, which have a lot of valuable information that can be exploited for decision support. However, these systems can handle any kind of data, including relational data. Apache Hadoop is one such system, and the Hive system allows SQL queries to be executed on top of the Hadoop system.

A number of companies provide software to optimize Hive query processing, including Cloudera and Hortonworks. Apache Spark is another popular Big Data system that supports SQL queries on data stored in files. Compressed file structures containing records with columns, such as Orc and Parquet, are increasingly used to store such log records, simplifying integration with SQL. Such file formats are discussed in more detail in Section 13.6.

Telegram: @uni_k

www.konkur.in

11.2.4 Data Lakes

While data warehouses pay a lot of attention to ensuring a common data schema to ease the job of querying the data, there are situations where organizations want to store data without paying the cost of creating a common schema and transforming data to the common schema. The term **data lake** is used to refer to a repository where data can be stored in multiple formats, including structured records and unstructured file formats. Unlike data warehouses, data lakes do not require up-front effort to preprocess data, but they do require more effort when creating queries. Since data may be stored in many different formats, querying tools also need to be quite flexible. Apache Hadoop and Apache Spark are popular tools for querying such data, since they support querying of both unstructured and structured data.

11.3 Online Analytical Processing

Data analysis often involves looking for patterns that arise when data values are grouped in "interesting" ways. As a simple example, summing credit hours for each department is a way to discover which departments have high teaching responsibilities. In a retail business, we might group sales by product, the date or month of the sale, the color or size of the product, or the profile (such as age group and gender) of the customer who bought the product.

11.3.1 Aggregation on Multidimensional Data

Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their item_name, color, and size, and that we have a relation *sales* with the schema.

$$sales\ (item_name, color, clothes_size, quantity)$$

Suppose that *item_name* can take on the values (skirt, dress, shirt, pants), *color* can take on the values (dark, pastel, white), *clothes_size* can take on values (small, medium, large), and *quantity* is an integer value representing the total number of items of a given {*item_name*, *color*, *clothes_size*}. An instance of the *sales* relation is shown in Figure 11.3.

Statistical analysis often requires grouping on multiple attributes. The attribute *quantity* of the *sales* relation is a measure attribute, since it measures the number of units sold, while *item_name*, *color*, and *clothes_size* are dimension attributes. (A more realistic version of the *sales* relation would have additional dimensions, such as time and sales location, and additional measures such as monetary value of the sale.)

To analyze the multidimensional data, a manager may want to see data laid out as shown in the table in Figure 11.4. The table shows total quantities for different combinations of *item_name* and *color*. The value of *clothes_size* is specified to be **all**, indicating that the displayed values are a summary across all values of *clothes_size* (i.e., we want to group the "small," "medium," and "large" items into one single group.

Telegram: @uni_k

www.konkur.in

dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3

Figure 11.3 An example of *sales* relation.

The table in Figure 11.4 is an example of a **cross-tabulation** (or **cross-tab**, for short), also referred to as a **pivot-table**. In general, a cross-tab is a table derived from a relation

Telegram: @uni_k

www.konkur.in

clothes_size **all**

item_name	*color*			
	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

Figure 11.4 Cross-tabulation of *sales* by *item_name* and *color*.

(say R), where values for one attribute (say A) form the column headers and values for another attribute (say B) form the row header. For example, in Figure 11.4, the attribute *color* corresponds to A (with values "dark," "pastel," and "white"), and the attribute *item_name* corresponds to B (with values "skirt," "dress," "shirt," and "pants").

Each cell in the pivot-table can be identified by (a_i, b_j), where a_i is a value for A and b_j a value for B. The values of the various cells in the pivot-table are derived from the relation R as follows: If there is at most one tuple in R with any (a_i, b_j) value, the value in the cell is derived from that single tuple (if any); for instance, it could be the value of one or more other attributes of the tuple. If there can be multiple tuples with an (a_i, b_j) value, the value in the cell must be derived by aggregation on the tuples with that value. In our example, the aggregation used is the sum of the values for attribute *quantity*, across all values for *clothes_size*, as indicated by "*clothes_size*: **all**" above the cross-tab in Figure 11.4. Thus, the value for cell (skirt, pastel) is 35, since there are three tuples in the *sales* table that meet that criteria, with values 11, 9, and 15.

In our example, the cross-tab also has an extra column and an extra row storing the totals of the cells in the row/column. Most cross-tabs have such summary rows and columns.

The generalization of a cross-tab, which is two-dimensional, to n dimensions can be visualized as an n-dimensional cube, called the **data cube**. Figure 11.5 shows a data cube on the *sales* relation. The data cube has three dimensions, *item_name*, *color*, and *clothes_size*, and the measure attribute is *quantity*. Each cell is identified by values for these three dimensions. Each cell in the data cube contains a value, just as in a cross-tab. In Figure 11.5, the value contained in a cell is shown on one of the faces of the cell; other faces of the cell are shown blank if they are visible. All cells contain values, even if they are not visible. The value for a dimension may be **all**, in which case the cell contains a summary over all values of that dimension, as in the case of cross-tabs.

The number of different ways in which the tuples can be grouped for aggregation can be large. In the example of Figure 11.5, there are 3 colors, 4 items, and 3 sizes resulting in a cube size of $3 \times 4 \times 3 = 36$. Including the summary values, we obtain a

Telegram: @uni_k

www.konkur.in

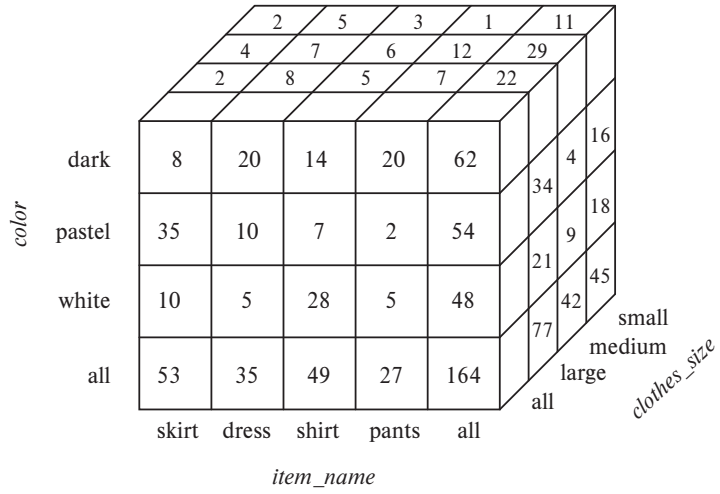


Figure 11.5 Three-dimensional data cube.

$4 \times 5 \times 4$ cube, whose size is 80. In fact, for a table with n dimensions, aggregation can be performed with grouping on each of the 2^n subsets of the n dimensions.[2]

An **online analytic processing** (OLAP) system allows a data analyst to look at different cross-tabs on the same data by *interactively* selecting the attributes in the cross-tab. Each cross-tab is a two-dimensional view on a multidimensional data cube. For instance, the analyst may select a cross-tab on *item_name* and *clothes_size* or a cross-tab on *color* and *clothes_size*. The operation of changing the dimensions used in a cross-tab is called **pivoting**.

OLAP systems allow an analyst to see a cross-tab on *item_name* and *color* for a fixed value of *clothes_size*, for example, large, instead of the sum across all sizes. Such an operation is referred to as **slicing**, since it can be thought of as viewing a slice of the data cube. The operation is sometimes called **dicing**, particularly when values for multiple dimensions are fixed.

When a cross-tab is used to view a multidimensional cube, the values of dimension attributes that are not part of the cross-tab are shown above the cross-tab. The value of such an attribute can be **all**, as shown in Figure 11.4, indicating that data in the cross-tab are a summary over all values for the attribute. Slicing/dicing simply consists of selecting specific values for these attributes, which are then displayed on top of the cross-tab.

OLAP systems permit users to view data at any desired level of granularity. The operation of moving from finer-granularity data to a coarser granularity (by means of aggregation) is called a **rollup**. In our example, starting from the data cube on the

[2]Grouping on the set of all n dimensions is useful only if the table may have duplicates.

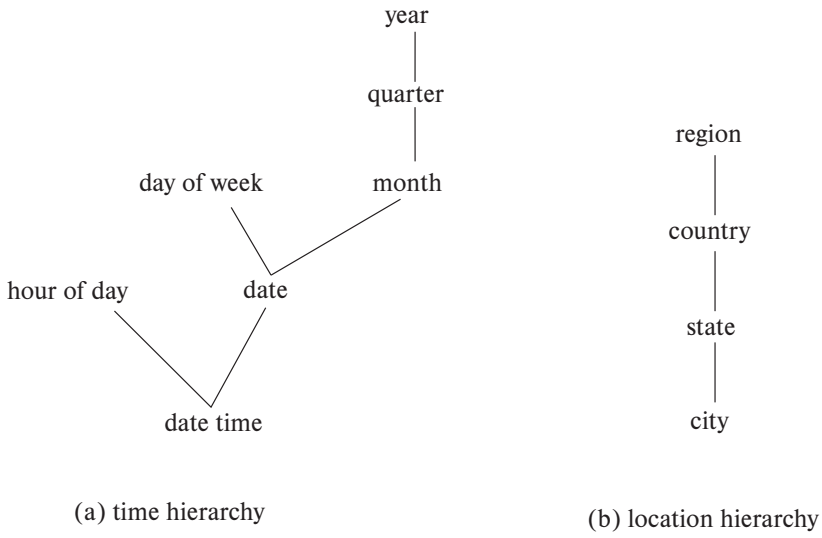
Telegram: @uni_k

www.konkur.in

sales table, we got our example cross-tab by rolling up on the attribute *clothes_size*. The opposite operation—that of moving from coarser-granularity data to finer-granularity data—is called a **drill down**. Finer-granularity data cannot be generated from coarse-granularity data; they must be generated either from the original data or from even finer-granularity summary data.

Analysts may wish to view a dimension at different levels of detail. For instance, consider an attribute of type **datetime** that contains a date and a time of day. Using time precise to a second (or less) may not be meaningful: An analyst who is interested in rough time of day may look at only the hour value. An analyst who is interested in sales by day of the week may map the date to a day of the week and look only at that. Another analyst may be interested in aggregates over a month, or a quarter, or for an entire year.

The different levels of detail for an attribute can be organized into a **hierarchy**. Figure 11.6a shows a hierarchy on the **datetime** attribute. As another example, Figure 11.6b shows a hierarchy on location, with the city being at the bottom of the hierarchy, state above it, country at the next level, and region being the top level. In our earlier example, clothes can be grouped by category (for instance, menswear or womenswear); *category* would then lie above *item_name* in our hierarchy on clothes. At the level of actual values, skirts and dresses would fall under the womenswear category and pants and shirts under the menswear category.



(a) time hierarchy

(b) location hierarchy

Figure 11.6 Hierarchies on dimensions.

Telegram: @uni_k

www.konkur.in

clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Figure 11.7 Cross-tabulation of *sales* with hierarchy on *item_name*.

An analyst may be interested in viewing sales of clothes divided as menswear and womenswear, and not interested in individual values. After viewing the aggregates at the level of womenswear and menswear, an analyst may *drill down the hierarchy* to look at individual values. An analyst looking at the detailed level may *drill up the hierarchy* and look at coarser-level aggregates. Both levels can be displayed on the same cross-tab, as in Figure 11.7.

11.3.2 Relational Representation of Cross-Tabs

A cross-tab is different from relational tables usually stored in databases, since the number of columns in the cross-tab depends on the actual data. A change in the data values may result in adding more columns, which is not desirable for data storage. However, a cross-tab view is desirable for display to users. It is straightforward to represent a cross-tab without summary values in a relational form with a fixed number of columns. A cross-tab with summary rows/columns can be represented by introducing a special value **all** to represent subtotals, as in Figure 11.8. The SQL standard actually uses the *null* value in place of **all**, but to avoid confusion with regular null values, we shall continue to use **all**.

Consider the tuples (skirt, **all**, **all**, 53) and (dress, **all**, **all**, 35). We have obtained these tuples by eliminating individual tuples with different values for *color* and *clothes_size*, and by replacing the value of *quantity* with an aggregate—namely, the sum of the quantities. The value **all** can be thought of as representing the set of all values for an attribute. Tuples with the value **all** for the *color* and *clothes_size* dimensions can be obtained by an aggregation on the *sales* relation with a **group by** on the column *item_name*. Similarly, a **group by** on *color*, *clothes_size* can be used to get the tuples with the value **all** for *item_name*, and a **group by** with no attributes (which can simply be omitted in SQL) can be used to get the tuple with value **all** for *item_name*, *color*, and *clothes_size*.

Telegram: @uni_k

www.konkur.in

skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	white	**all**	28
shirt	**all**	**all**	49
pants	dark	**all**	20
pants	pastel	**all**	2
pants	white	**all**	5
pants	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Figure 11.8 Relational representation of the data in Figure 11.4.

Hierarchies can also be represented by relations. For example, the fact that skirts and dresses fall under the womenswear category and the pants and shirts under the menswear category can be represented by a relation *itemcategory* (*item_name*, *category*). This relation can be joined with the *sales* relation to get a relation that includes the category for each item. Aggregation on this joined relation allows us to get a cross-tab with hierarchy. As another example, a hierarchy on city can be represented by a single relation *city_hierarchy* (ID, *city*, *state*, *country*, *region*), or by multiple relations, each mapping values in one level of the hierarchy to values at the next level. We assume here that cities have unique identifiers, stored in the attribute ID, to avoid confusing between two cities with the same name, for example, the Springfield in Missouri and the Springfield in Illinois.

11.3.3 OLAP in SQL

Analysts using OLAP systems need answers to multiple aggregates to be generated interactively, without having to wait for multiple minutes or hours. This led initially to the development of specialized systems for OLAP (see Note 11.1 on page 535). Many database systems now implement OLAP along with SQL constructs to express OLAP queries. As we saw in Section 5.5.3, several implementations, such as Microsoft

Telegram: @uni_k

www.konkur.in

SQL Server and Oracle, support a **pivot** clause that allows creation of cross-tabs. Given the *sales* relation from Figure 11.3, the query:

```
select *
from sales
pivot (
        sum(quantity)
        for color in ('dark','pastel','white')
)
order by item_name;
```

returns the cross-tab shown in Figure 11.9. Note that the **for** clause within the **pivot** clause specifies the *color* values that appear as attribute names in the pivot result. The attribute *color* itself does not appear in the result, although all other attributes are retained, except that the values for the newly created attributes are specified to come from the attribute *quantity*. In case more than one tuple contributes values to a given cell, the aggregate operation within the **pivot** clause specifies how the values should be combined. In the above example, the *quantity* values are summed up.

Note that the pivot clause by itself does not compute the subtotals we saw in the pivot table from Figure 11.4. However, we can first generate the relational representation shown in Figure 11.8, using a cube operation, as outlined shortly, and then apply the pivot clause on that representation to get an equivalent result. In this case, the value **all** must also be listed in the **for** clause, and the **order by** clause needs to be modified to order **all** at the end.

The data in a data cube cannot be generated by a single SQL query if we use only the basic **group by** constructs, since aggregates are computed for several different groupings

dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3

Figure 11.9 Result of SQL pivot operation on the *sales* relation of Figure 11.3.

Telegram: @uni_k

www.konkur.in

multidimensional OLAP (MOLAP)

relational OLAP (ROLAP)

hybrid OLAP (HOLAP)

Telegram: @uni_k

www.konkur.in

of the dimension attributes. Using only the basic **group by** construct, we would have to write many separate SQL queries and combine them using a union operation. SQL supports special syntax to allow multiple group by operations to be specified concisely.

As we saw in Section 5.5.4, SQL supports generalizations of the **group by** construct to perform the **cube** and **rollup** operations. The **cube** and **rollup** constructs in the **group by** clause allow multiple **group by** queries to be run in a single query with the result returned as a single relation in a style similar to that of the relation of Figure 11.8.

Consider again our retail shop example and the relation:

sales (*item_name*, *color*, *clothes_size*, *quantity*)

If we want to generate the entire data cube using individual group by queries, we have to write a separate query for each of the following eight sets of group by attributes:

{ (*item_name*, *color*, *clothes_size*), (*item_name*, *color*), (*item_name*, *clothes_size*), (*color*, *clothes_size*), (*item_name*), (*color*), (*clothes_size*), () }

where () denotes an empty **group by** list.

As we saw in Section 5.5.4, the **cube** construct allows us to accomplish this in one query:

select *item_name*, color, *clothes_size*, **sum**(*quantity*)
from *sales*
group by cube(*item_name*, *color*, *clothes_size*);

The preceding query produces a relation whose schema is:

(*item_name*, *color*, *clothes_size*, **sum**(*quantity*))

So that the result of this query is indeed a relation, tuples in the result contain *null* as the value of those attributes not present in a particular grouping. For example, tuples produced by grouping on *clothes_size* have a schema (*clothes_size*, **sum**(*quantity*)). They are converted to tuples on (*item_name*, *color*, *clothes_size*, **sum**(*quantity*)) by inserting *null* for *item_name* and *color*.

Data cube relations are often very large. The cube query above, with 3 possible colors, 4 possible item names, and 3 sizes, has 80 tuples. The relation of Figure 11.8 is generated by doing a **group by cube** on *item_name* and *color*, with an extra column specified in the **select** clause showing **all** for *clothes_size*.

To generate that relation in SQL, we substitute **all** for *null* using the **grouping** function, as we saw earlier in Section 5.5.4. The **grouping** function distinguishes those nulls generated by OLAP operations from "normal" nulls actually stored in the database or arising from an outer join. Recall that the **grouping** function returns 1 if its argument

Telegram: @uni_k

www.konkur.in

is a null value generated by a **cube** or **rollup** and 0 otherwise. We may then operate on the result of a call to **grouping** using **case** expressions to replace OLAP-generated nulls with **all**. Then the relation in Figure 11.8, with occurrences of *null* replaced by **all**, can be computed by the query:

```
select case when grouping(item_name) = 1 then 'all'
            else item_name end as item_name,
       case when grouping(color) = 1 then 'all'
            else color end as color,
       'all' as clothes_size, sum(quantity) as quantity
from sales
group by cube(item_name, color);
```

The **rollup** construct is the same as the **cube** construct except that **rollup** generates fewer **group by** queries. We saw that **group by cube** (*item_name*, *color*, *clothes_size*) generated all eight ways of forming a **group by** query using some (or all or none) of the attributes. In:

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name, color, clothes_size);
```

the clause **group by rollup**(*item_name*, *color*, *clothes_size*) generates only four groupings:

$$\{ (item_name, color, clothes_size), (item_name, color), (item_name), () \}$$

Notice that the order of the attributes in the **rollup** makes a difference; the last attribute (*clothes_size*, in our example) appears in only one grouping, the penultimate (second last) attribute in two groupings, and so on, with the first attribute appearing in all groups but one (the empty grouping).

Why might we want the specific groupings that are used in **rollup**? These groups are of frequent practical interest for hierarchies (as in Figure 11.6, for example). For the location hierarchy (*Region, Country, State, City*), we may want to group by *Region* to get sales by region. Then we may want to "drill down" to the level of countries within each region, which means we would group by *Region, Country*. Drilling down further, we may wish to group by *Region, Country, State* and then by *Region, Country, State, City*. The **rollup** construct allows us to specify this sequence of drilling down for further detail.

As we saw earlier in Section 5.5.4, multiple **rollup**s and **cube**s can be used in a single **group by** clause. For instance, the following query:

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name), rollup(color, clothes size);
```

Telegram: @uni_k

www.konkur.in

generates the groupings:

{ (*item_name*, *color*, *clothes_size*), (*item_name*, *color*), (*item_name*), (*color*, *clothes_size*), (*color*), () }

To understand why, observe that **rollup**(*item_name*) generates two groupings, {(*item_name*), ()}, and **rollup**(*color*, *clothes_size*) generates three groupings, {(*color*, *clothes_size*), (*color*), () }. The Cartesian product of the two gives us the six groupings shown.

Neither the **rollup** nor the **cube** clause gives complete control on the groupings that are generated. For instance, we cannot use them to specify that we want only groupings {(*color*, *clothes_size*), (*clothes_size*, *item_name*)}. Such restricted groupings can be generated by using the **grouping sets** construct, in which one can specify the specific list of groupings to be used. To obtain only groupings {(*color*, *clothes_size*), (*clothes_size*, *item_name*)}, we would write:

select *item_name*, *color*, *clothes_size*, **sum**(*quantity*)
from *sales*
group by grouping sets ((*color*, *clothes_size*), (*clothes_size*, *item_name*));

Specialized languages have been developed for querying multidimensional OLAP schemas, which allow some common tasks to be expressed more easily than with SQL. These include the MDX and DAX query languages developed by Microsoft.

11.3.4 Reporting and Visualization Tools

Report generators are tools to generate human-readable summary reports from a database. They integrate querying the database with the creation of formatted text and summary charts (such as bar or pie charts). For example, a report may show the total sales in each of the past 2 months for each sales region.

The application developer can specify report formats by using the formatting facilities of the report generator. Variables can be used to store parameters such as the month and the year and to define fields in the report. Tables, graphs, bar charts, or other graphics can be defined via queries on the database. The query definitions can make use of the parameter values stored in the variables.

Once we have defined a report structure on a report-generator facility, we can store it and can execute it at any time to generate a report. Report-generator systems provide a variety of facilities for structuring tabular output, such as defining table and column headers, displaying subtotals for each group in a table, automatically splitting long tables into multiple pages, and displaying subtotals at the end of each page.

Figure 11.10 is an example of a formatted report. The data in the report are generated by aggregation on information about orders.

Report-generation tools are available from a variety of vendors, such as SAP Crystal Reports and Microsoft (SQL Server Reporting Services). Several application suites, such as Microsoft Office, provide a way of embedding formatted query results from

Telegram: @uni_k

www.konkur.in

Acme Supply Company, Inc.
Quarterly Sales Report

Period: Jan. 1 to March 31, 2009

Region	Category	Sales	Subtotal
North	Computer Hardware	1,000,000	
	Computer Software	500,000	
	All categories		1,500,000
South	Computer Hardware	200,000	
	Computer Software	400,000	
	All categories		600,000
		Total Sales	2,100,000

Figure 11.10 A formatted report.

a database directly into a document. Chart-generation facilities provided by Crystal Reports or by spreadsheets such as Excel can be used to access data from databases and to generate tabular depictions of data or graphical depictions using charts or graphs. Such charts can be embedded within text documents. The charts are created initially from data generated by executing queries against the database; the queries can be re-executed and the charts regenerated when required, to generate a current version of the overall report.

Techniques for **data visualization**, that is, graphical representation of data, that go beyond the basic chart types, are very important for data analysis. Data-visualization systems help users to examine large volumes of data and to detect patterns visually. Visual displays of data—such as maps, charts, and other graphical representations—allow data to be presented compactly to users. A single graphical screen can encode as much information as a far larger number of text screens.

For example, if the user wants to find out whether the occurrence of a disease is correlated to the locations of the patients, the locations of patients can be encoded in a special color—say, red—on a map. The user can then quickly discover locations where problems are occurring. The user may then form hypotheses about why problems are occurring in those locations and may verify the hypotheses quantitatively against the database.

As another example, information about values can be encoded as a color and can be displayed with as little as one pixel of screen area. To detect associations between pairs of items, we can use a two-dimensional pixel matrix, with each row and each column representing an item. The percentage of transactions that buy both items can be encoded by the color intensity of the pixel. Items with high association will show up as bright pixels in the screen—easy to detect against the darker background.

Telegram: @uni_k

www.konkur.in

In recent years a number of tools have been developed for web-based data visualization and for the creation of dashboards that display multiple charts showing key organizational information. These include Tableau (www.tableau.com), FusionCharts (www.fusioncharts.com), plotly (plot.ly), Datawrapper (www.datawrapper.de), and Google Charts (developers.google.com/chart), among others. Since their display is based on HTML and JavaScript, they can be used on a wide variety of browsers and on mobile devices.

Interaction is a key element of visualization. For example, a user can "drill down" into areas of interest, such as moving from an aggregate view showing the total sales across an entire year to the monthly sales figures for a particular year. Analysts may wish to interactively add selection conditions to visualize subsets of data. Data visualization tools such as Tableau offer a rich set of features for interactive visualization.

11.4 Data Mining

The term **data mining** refers loosely to the process of analyzing large databases to find useful patterns. Like knowledge discovery in artificial intelligence (also called machine learning) or statistical analysis, data mining attempts to discover rules and patterns from data. However, data mining differs from traditional machine learning and statistics in that it deals with large volumes of data, stored primarily on disk. Today, many machine-learning algorithms also work on very large volumes of data, blurring the distinction between data mining and machine learning. Data-mining techniques form part of the process of **knowledge discovery in databases** (**KDD**).

Some types of knowledge discovered from a database can be represented by a set of **rules**. The following is an example of a rule, stated informally: "Young women with annual incomes greater than $50,000 are the most likely people to buy small sports cars." Of course such rules are not universally true and have degrees of "support" and "confidence," as we shall see. Other types of knowledge are represented by equations relating different variables to each other. More generally, knowledge discovered by applying machine-learning techniques on past instances in a database is represented by a **model**, which is then used for predicting outcomes for new instances. Features or attributes of instances are inputs to the model, and the output of a model is a prediction.

There are a variety of possible types of patterns that may be useful, and different techniques are used to find different types of patterns. We shall study a few examples of patterns and see how they may be automatically derived from a database.

Usually there is a manual component to data mining, consisting of preprocessing data to a form acceptable to the algorithms and post-processing of discovered patterns to find novel ones that could be useful. There may also be more than one type of pattern that can be discovered from a given database, and manual interaction may be needed to pick useful types of patterns. For this reason, data mining is really a semiautomatic process in real life. However, in our description we concentrate on the automatic aspect of mining.

Telegram: @uni_k

www.konkur.in

11.4.1 Types of Data-Mining Tasks

The most widely used applications of data mining are those that require some sort of **prediction**. For instance, when a person applies for a credit card, the credit-card company wants to predict if the person is a good credit risk. The prediction is to be based on known attributes of the person, such as age, income, debts, and past debt-repayment history. Rules for making the prediction are derived from the same attributes of past and current credit-card holders, along with their observed behavior, such as whether they defaulted on their credit-card dues. Other types of prediction include predicting which customers may switch over to a competitor (these customers may be offered special discounts to tempt them not to switch), predicting which people are likely to respond to promotional mail ("junk mail"), or predicting what types of phone calling-card usage are likely to be fraudulent.

Another class of applications looks for **associations**, for instance, books that tend to be bought together. If a customer buys a book, an online bookstore may suggest other associated books. If a person buys a camera, the system may suggest accessories that tend to be bought along with cameras. A good salesperson is aware of such patterns and exploits them to make additional sales. The challenge is to automate the process. Other types of associations may lead to discovery of causation. For instance, discovery of unexpected associations between a newly introduced medicine and cardiac problems led to the finding that the medicine may cause cardiac problems in some people. The medicine was then withdrawn from the market.

Associations are an example of **descriptive patterns**. **Clusters** are another example of such patterns. For example, over a century ago a cluster of typhoid cases was found around a well, which led to the discovery that the water in the well was contaminated and was spreading typhoid. Detection of clusters of disease remains important even today.

11.4.2 Classification

Abstractly, the **classification** problem is this: Given that items belong to one of several classes, and given past instances (called **training instances**) of items along with the classes to which they belong, the problem is to predict the class to which a new item belongs. The class of the new instance is not known, so other attributes of the instance must be used to predict the class.

As an example, suppose that a credit-card company wants to decide whether or not to give a credit card to an applicant. The company has a variety of information about the person, such as her age, educational background, annual income, and current debts, that it can use for making a decision.

To make the decision, the company assigns a credit worthiness level of excellent, good, average, or bad to each of a sample set of current or past customers according to each customer's payment history. These instances form the set of training instances.

Then, the company attempts to learn rules or models that classify the credit-worthiness of a new applicant as excellent, good, average, or bad, on the basis of the

Telegram: @uni_k

www.konkur.in

information about the person, other than the actual payment history (which is unavailable for new customers). There are a number of techniques for classification, and we outline a few of them in this section.

11.4.2.1 Decision-Tree Classifiers

Decision-tree classifiers are a widely used technique for classification. As the name suggests, **decision-tree classifiers** use a tree; each leaf node has an associated class, and each internal node has a predicate (or more generally, a function) associated with it. Figure 11.11 shows an example of a decision tree. To keep the example simple, we use just two attributes: education level (highest degree earned) and income.

To classify a new instance, we start at the root and traverse the tree to reach a leaf; at an internal node we evaluate the predicate (or function) on the data instance to find which child to go to. The process continues until we reach a leaf node. For example, if the degree level of a person is masters, and the person's income is 40K, starting from the root we follow the edge labeled "masters," and from there the edge labeled "25K to 75K," to reach a leaf. The class at the leaf is "good," so we predict that the credit risk of that person is good.

There are a number of techniques for building decision-tree classifiers from a given training set. We omit details, but you can learn more details from the references provided in the Further Reading section.

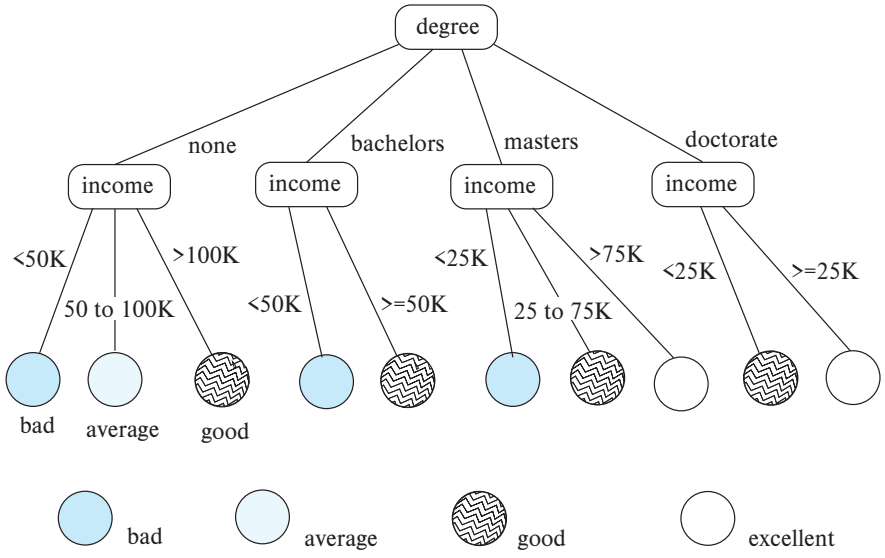


Figure 11.11 Classification tree.

Telegram: @uni_k

www.konkur.in

11.4.2.2 Bayesian Classifiers

Bayesian classifiers find the distribution of attribute values for each class in the training data; when given a new instance d, they use the distribution information to estimate, for each class c_j, the probability that instance d belongs to class c_j, denoted by $p(c_j|d)$, in a manner outlined here. The class with maximum probability becomes the predicted class for instance d.

To find the probability $p(c_j|d)$ of instance d being in class c_j, Bayesian classifiers use **Bayes' theorem**, which says:

$$p(c_j|d) = \frac{p(d|c_j)p(c_j)}{p(d)}$$

where $p(d|c_j)$ is the probability of generating instance d given class c_j, $p(c_j)$ is the probability of occurrence of class c_j, and $p(d)$ is the probability of instance d occurring. Of these, $p(d)$ can be ignored since it is the same for all classes. $p(c_j)$ is simply the fraction of training instances that belong to class c_j.

For example, let us consider a special case where only one attribute, *income*, is used for classification, and suppose we need to classify a person whose income is 76,000. We assume that income values are broken up into buckets, and we assume that the bucket containing 76,000 contains values in the range (75,000, 80,000). Suppose among instances of class *excellent*, the probability of income being in (75,000, 80,000) is 0.1, while among instances of class *good*, the probability of income being in (75,000, 80,000) is 0.05. Suppose also that overall 0.1 fraction of people are classified as *excellent*, and 0.3 are classified as *good*. Then, $p(d|c_j)p(c_j)$ for class *excellent* is .01, while for class *good*, it is 0.015. The person would therefore be classified in class *good*.

In general, multiple attributes need to be considered for classification. Then, finding $p(d|c_j)$ exactly is difficult, since it requires the distribution of instances of c_j, across all combinations of values for the attributes used for classification. The number of such combinations (for example of income buckets, with degree values and other attributes) can be very large. With a limited training set used to find the distribution, most combinations would not have even a single training set matching them, leading to incorrect classification decisions. To avoid this problem, as well as to simplify the task of classification, **naive Bayesian classifiers** assume attributes have independent distributions and thereby estimate:

$$p(d|c_j) = p(d_1|c_j) * p(d_2|c_j) * \cdots * p(d_n|c_j)$$

That is, the probability of the instance d occurring is the product of the probability of occurrence of each of the attribute values d_i of d, given the class is c_j.

The probabilities $p(d_i|c_j)$ derive from the distribution of values for each attribute i, for each class c_j. This distribution is computed from the training instances that belong to each class c_j; the distribution is usually approximated by a histogram. For instance, we may divide the range of values of attribute i into equal intervals, and store the fraction of instances of class c_j that fall in each interval. Given a value d for attribute i, the

Telegram: @uni_k

www.konkur.in

value of $p(d_i|c_j)$ is simply the fraction of instances belonging to class c_j that fall in the interval to which d_i belongs.

11.4.2.3 Support Vector Machine Classifiers

The **Support Vector Machine** (SVM) is a type of classifier that has been found to give very accurate classification across a range of applications. We provide some basic information about Support Vector Machine classifiers here; see the references in the bibliographical notes for further information.

Support Vector Machine classifiers can best be understood geometrically. In the simplest case, consider a set of points in a two-dimensional plane, some belonging to class A, and some belonging to class B. We are given a training set of points whose class (A or B) is known, and we need to build a classifier of points using these training points. This situation is illustrated in Figure 11.12, where the points in class A are denoted by X marks, while those in class B are denoted by O marks.

Suppose we can draw a line on the plane, such that all points in class A lie to one side and all points in line B lie to the other. Then, the line can be used to classify new points, whose class we don't already know. But there may be many possible such lines that can separate points in class A from points in class B. A few such lines are shown in Figure 11.12. The Support Vector Machine classifier chooses the line whose distance from the nearest point in either class (from the points in the training dataset) is maximum. This line (called the *maximum margin line*) is then used to classify other points into class A or B, depending on which side of the line they lie on. In Figure 11.12, the maximum margin line is shown in bold, while the other lines are shown as dashed lines.

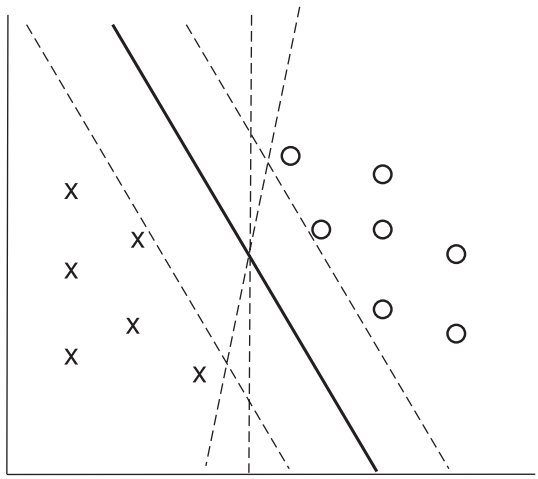


Figure 11.12 Example of a Support Vector Machine classifier.

Telegram: @uni_k

www.konkur.in

The preceding intuition can be generalized to more than two dimensions, allowing multiple attributes to be used for classification; in this case, the classifier finds a dividing plane, not a line. Further, by first transforming the input points using certain functions, called *kernel functions*, Support Vector Machine classifiers can find nonlinear curves separating the sets of points. This is important for cases where the points are not separable by a line or plane. In the presence of noise, some points of one class may lie in the midst of points of the other class. In such cases, there may not be any line or meaningful curve that separates the points in the two classes; then, the line or curve that most accurately divides the points into the two classes is chosen.

Although the basic formulation of Support Vector Machines is for binary classifiers, i.e., those with only two classes, they can be used for classification into multiple classes as follows: If there are N classes, we build N classifiers, with classifier i performing a binary classification, classifying a point either as in class i or not in class i. Given a point, each classifier i also outputs a value indicating how related a given point is to class i. We then apply all N classifiers on a given point and choose the class for which the relatedness value is the highest.

11.4.2.4 Neural Network Classifiers

Neural-net classifiers use the training data to train artificial neural nets. There is a large body of literature on neural nets; we do not provide details here, but we outline a few key properties of neural network classifiers.

Neural networks consist of several layers of "neurons," each of which are connected to neurons in the preceding layer. An input instance of the problem is fed to the first layer; neurons at each layer are "activated" based on some function applied to the inputs at the preceding layer. The function applied at each neuron computes a weighted combination of the activations of the input neurons and generates an output based on the weighted combination. The activation of a neuron in one layer thus affects the activation of neurons in the next layer. The final output layer typically has one neuron corresponding to each class of the classification problem being addressed. The neuron with maximum activation for a given input decides the predicted class for that input.

Key to the success of a neural network is the weights used in the computation described above. These weights are learned, based on training data. They are initially set to some default value, and then training data are used to learn the weights. Training is typically done by applying each input to the current state of the neural network and checking if the prediction is correct. If not, a *backpropagation algorithm* is used to tweak the weights of the neurons in the network, to bring the prediction closer to the correct one for the current input. Repeating this process results in a trained neural network, which can then be used for classification on new inputs.

In recent years, neural networks have achieved a great degree of success for tasks which were earlier considered very hard, such as vision (e.g., recognition of objects in images), speech recognition, and natural language translation. A simple example of a vision task is that of identifying the species, such as cat or dog, given an image of

Telegram: @uni_k

www.konkur.in

an animal; such problems are basically classification problems. Other examples include identifying object occurrences in an image and assigning a class label to each identified object.

Deep neural networks, which are neural networks with a large number of layers, have proven very successful at such tasks, if given a very large number of training instances. The term **deep learning** refers to the machine-learning techniques that create such deep neural networks, and train them on very large numbers of training instances.

11.4.3 Regression

Regression deals with the prediction of a value, rather than a class. Given values for a set of variables, $X_1, X_2, \ldots, X_n$, we wish to predict the value of a variable Y. For instance, we could treat the level of education as a number and income as another number, and, on the basis of these two variables, we wish to predict the likelihood of default, which could be a percentage chance of defaulting, or the amount involved in the default.

One way is to infer coefficients $a_0, a_1, a_2, \ldots, a_n$ such that:

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \cdots + a_n * X_n$$

Finding such a linear polynomial is called **linear regression**. In general, we wish to find a curve (defined by a polynomial or other formula) that fits the data; the process is also called **curve fitting**.

The fit may be only approximate, because of noise in the data or because the relationship is not exactly a polynomial, so regression aims to find coefficients that give the best possible fit. There are standard techniques in statistics for finding regression coefficients. We do not discuss these techniques here, but the bibliographical notes provide references.

11.4.4 Association Rules

Retail shops are often interested in associations between different items that people buy. Examples of such associations are:

- Someone who buys bread is quite likely also to buy milk.
- A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.

Association information can be used in several ways. When a customer buys a particular book, an online shop may suggest associated books. A grocery shop may decide to place bread close to milk, since they are often bought together, to help shoppers finish their task faster. Or, the shop may place them at opposite ends of a row and place other associated items in between to tempt people to buy those items as well as the shoppers walk from one end of the row to the other. A shop that offers discounts on one associ-

Telegram: @uni_k

www.konkur.in

ated item may not offer a discount on the other, since the customer will probably buy the other anyway.

An example of an association rule is:

$$bread \Rightarrow milk$$

In the context of grocery-store purchases, the rule says that customers who buy bread also tend to buy milk with a high probability. An association rule must have an associated **population**: The population consists of a set of **instances**. In the grocery-store example, the population may consist of all grocery-store purchases; each purchase is an instance. In the case of a bookstore, the population may consist of all people who made purchases, regardless of when they made a purchase. Each customer is an instance. In the bookstore example, the analyst has decided that when a purchase is made is not significant, whereas for the grocery-store example, the analyst may have decided to concentrate on single purchases, ignoring multiple visits by the same customer.

Rules have an associated *support*, as well as an associated *confidence*. These are defined in the context of the population:

- **Support** is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.

  For instance, suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule:

  $$milk \Rightarrow screwdrivers$$

  is low. The rule may not even be statistically significant—perhaps there was only a single purchase that included both milk and screwdrivers. Businesses are usually not interested in rules that have low support, since they involve few customers and are not worth bothering about.

  On the other hand, if 50 percent of all purchases involve milk and bread, then support for rules involving bread and milk (and no other item) is relatively high, and such rules may be worth attention. Exactly what minimum degree of support is considered desirable depends on the application.

- **Confidence** is a measure of how often the consequent is true when the antecedent is true. For instance, the rule:

  $$bread \Rightarrow milk$$

  has a confidence of 80 percent if 80 percent of the purchases that include bread also include milk. A rule with a low confidence is not meaningful. In business applications, rules usually have confidences significantly less than 100 percent, whereas in other domains, such as in physics, rules may have high confidences.

  Note that the confidence of $bread \Rightarrow milk$ may be very different from the confidence of $milk \Rightarrow bread$, although both have the same support.

Telegram: @uni_k

www.konkur.in

11.4.5 Clustering

Intuitively, clustering refers to the problem of finding clusters of points in the given data. The problem of **clustering** can be formalized from distance metrics in several ways. One way is to phrase it as the problem of grouping points into k sets (for a given k) so that the average distance of points from the *centroid* of their assigned cluster is minimized. [3] Another way is to group points so that the average distance between every pair of points in each cluster is minimized. There are other definitions too; see the bibliographical notes for details. But the intuition behind all these definitions is to group similar points together in a single set.

Another type of clustering appears in classification systems in biology. (Such classification systems do not attempt to *predict* classes; rather they attempt to cluster related items together.) For instance, leopards and humans are clustered under the class mammalia, while crocodiles and snakes are clustered under reptilia. Both mammalia and reptilia come under the common class chordata. The clustering of mammalia has further subclusters, such as carnivora and primates. We thus have **hierarchical clustering**. Given characteristics of different species, biologists have created a complex hierarchical clustering scheme grouping related species together at different levels of the hierarchy.

The statistics community has studied clustering extensively. Database research has provided scalable clustering algorithms that can cluster very large datasets (that may not fit in memory).

An interesting application of clustering is to predict what new movies (or books or music) a person is likely to be interested in on the basis of:

1. The person's past preferences in movies.
2. Other people with similar past preferences.
3. The preferences of such people for new movies.

One approach to this problem is as follows: To find people with similar past preferences we create clusters of people based on their preferences for movies. The accuracy of clustering can be improved by previously clustering movies by their similarity, so even if people have not seen the same movies, if they have seen similar movies they would be clustered together. We can repeat the clustering, alternately clustering people, then movies, then people, and so on until we reach an equilibrium. Given a new user, we find a cluster of users most similar to that user, on the basis of the user's preferences for movies already seen. We then predict movies in movie clusters that are popular with that user's cluster as likely to be interesting to the new user. In fact, this problem

[3] The centroid of a set of points is defined as a point whose coordinate on each dimension is the average of the coordinates of all the points of that set on that dimension. For example, in two dimensions, the centroid of a set of points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ is given by $\left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right)$.

Telegram: @uni_k

www.konkur.in

is an instance of *collaborative filtering*, where users collaborate in the task of filtering information to find information of interest.

11.4.6 Text Mining

Text mining applies data-mining techniques to textual documents. There are a number of different text mining tasks. One such task is **sentiment analysis**. For example, suppose a company wishes to find out how users have reacted to a new product. There are typically a large number of product reviews on the web—for example, reviews by different users on e-commerce platforms. Reading each review to find out reactions is not practical for a human. Instead, the company may analyze reviews to find the *sentiment* of the reviews of the product; the sentiment could be positive, negative, or neutral. The occurrence of specific words such as excellent, good, awesome, beautiful, and so on are correlated with a positive sentiment, while words such as awful, average, worthless, poor quality, and so on are correlated with a negative sentiment. Sentiment analysis techniques can be used to analyze the reviews and come up with an overall score reflecting the broad sense of the reviews.

Another task is **information extraction**, which creates structured information from unstructured textual descriptions, or semi-structured data such as tabular displays of data in documents. A key subtask of this process is **entity recognition**, that is, the task of identifying mentions of entities in text and disambiguating them. For example, an article may mention the name Michael Jordan. There are at least two famous people named Michael Jordan: one was a basketball player, while the other is a professor who is a well known machine-learning expert. **Disambiguation** is the process of figuring out which of these two is being referred to in a particular article, and it can be done based on the article context; in this case, an occurrence of the name Michael Jordan in a sports article probably refers to the basketball player, while an occurrence in a machine-learning paper probably refers to the professor. After entity recognition, other techniques may be used to learn attributes of entities and to learn relationships between entities.

Information extraction can be used in many ways. For example, it can be used to analyze customer support conversations or reviews posted on social media, to judge customer satisfaction, and to decide when intervention is needed to retain customers. Service providers may want to know what aspect of the service such as pricing, quality, hygiene, or behavior of the person providing the service, a review was positive or negative about; information extraction techniques can be used to infer what aspect an article or a part of an article is about, and to infer the associated sentiment. Attributes such as the location of service can also be extracted and are important for taking corrective action.

Information extracted from the enormous collection of documents and other resources on the web can be valuable for many tasks. Such extracted information can be represented in a graph, called a **knowledge graph**, which we outlined in Section 8.1.4. Such knowledge graphs are used by web search engines to generate more meaningful answers to user queries.

Telegram: @uni_k

www.konkur.in

11.5 Summary

- Data analytics systems analyze online data collected by transaction-processing systems, along with data collected from other sources, to help people make business decisions. Decision-support systems come in various forms, including OLAP systems and data-mining systems.
- Data warehouses help gather and archive important operational data. Warehouses are used for decision support and analysis on historical data, for instance, to predict trends. Data cleansing from input data sources is often a major task in data warehousing. Warehouse schemas tend to be multidimensional, involving one or a few very large fact tables and several much smaller dimension tables.
- Online analytical processing (OLAP) tools help analysts view data summarized in different ways, so that they can gain insight into the functioning of an organization.
    - OLAP tools work on multidimensional data, characterized by dimension attributes and measure attributes.
    - The data cube consists of multidimensional data summarized in different ways. Precomputing the data cube helps speed up queries on summaries of data.
    - Cross-tab displays permit users to view two dimensions of multidimensional data at a time, along with summaries of the data.
    - Drill down, rollup, slicing, and dicing are among the operations that users perform with OLAP tools.
- The SQL standard provides a variety of operators for data analysis, including **cube**, **rollup**, and **pivot** operations.
- Data mining is the process of semiautomatically analyzing large databases to find useful patterns. There are a number of applications of data mining, such as prediction of values based on past examples, finding of associations between purchases, and automatic clustering of people and movies.
- Classification deals with predicting the class of test instances by using attributes of the test instances, based on attributes of training instances, and the actual class of training instances. There are several types of classifiers, such as:
    - Decision-tree classifiers, which perform classification by constructing a tree based on training instances with leaves having class labels.
    - Bayesian classifiers, which are based on probability theory.
    - The support vector machine is another widely used classification technique.
    - Neural networks, and in particular deep learning, has been very successful in classification and related tasks in the context of vision, speech recognition, and language understanding and translation.

Telegram: @uni_k

www.konkur.in

- Association rules identify items that co-occur frequently, for instance, items that tend to be bought by the same customer. Correlations look for deviations from expected levels of association.
- Other types of data mining include clustering and text mining.

Review Terms

- Decision-support systems
- Business intelligence
- Data warehousing
  - Gathering data
  - Source-driven architecture
  - Destination-driven architecture
  - Data cleansing
  - Extract, transform, load (ETL)
  - Extract, load, transform (ELT)
- Warehouse schemas
  - Fact table
  - Dimension tables
  - Star schema
  - Snowflake schema
- Column-oriented storage
- Online analytical processing (OLAP)
- Multidimensional data
  - Measure attributes
  - Dimension attributes
  - Hierarchy
  - Cross-tabulation / Pivoting
  - Data cube
  - Rollup and drill down
  - SQL **group by cube**, **group by rollup**
- Data visualization
- Data mining
- Prediction
- Classification
  - Training data
  - Test data
- Decision-tree classifiers
- Bayesian classifiers
  - Bayes' theorem
  - Naive Bayesian classifiers
- Support Vector Machine (SVM)
- Regression
- Neural-networks
- Deep learning
- Association rules
- Clustering
- Text mining
  - Sentiment analysis
  - Information extraction
  - Named entity recognition
  - Knowledge graph

Telegram: @uni_k

www.konkur.in

Practice Exercises

11.1 Describe benefits and drawbacks of a source-driven architecture for gathering of data at a data warehouse, as compared to a destination-driven architecture.

11.2 Draw a diagram that shows how the *classroom* relation of our university example as shown in Appendix A would be stored under a column-oriented storage structure.

11.3 Consider the *takes* relation. Write an SQL query that computes a cross-tab that has a column for each of the years 2017 and 2018, and a column for **all**, and one row for each course, as well as a row for **all**. Each cell in the table should contain the number of students who took the corresponding course in the corresponding year, with column **all** containing the aggregate across all years, and row **all** containing the aggregate across all courses.

11.4 Consider the data warehouse schema depicted in Figure 11.2. Give an SQL query to summarize sales numbers and price by store and date, along with the hierarchies on store and date.

11.5 Classification can be done using *classification rules*, which have a *condition*, a *class*, and a *confidence*; the confidence is the percentage of the inputs satisfying the condition that fall in the specified class.

For example, a classification rule for credit ratings may have a condition that salary is between $30,000 and $50,000, and education level is graduate, with the credit rating class of *good*, and a confidence of 80%. A second rule may have a condition that salary is between $30,000 and $50,000, and education level is high-school, with the credit rating class of *satisfactory*, and a confidence of 80%. A third rule may have a condition that salary is above $50,001, with the credit rating class of *excellent*, and confidence of 90%. Show a decision tree classifier corresponding to the above rules.

Show how the decision tree classifier can be extended to record the confidence values.

11.6 Consider a classification problem where the classifier predicts whether a person has a particular disease. Suppose that 95% of the people tested do not suffer from the disease. Let *pos* denote the fraction of *true positives*, which is 5% of the test cases, and let *neg* denote the fraction of *true negatives*, which is 95% of the test cases. Consider the following classifiers:

- Classifier C_1, which always predicts negative (a rather useless classifier, of course).
- Classifier C_2, which predicts positive in 80% of the cases where the person actually has the disease but also predicts positive in 5% of the cases where the person does not have the disease.

Telegram: @uni_k

www.konkur.in

- Classifier C_3, which predicts positive in 95% of the cases where the person actually has the disease but also predicts positive in 20% of the cases where the person does not have the disease.

For each classifier, let *t_pos* denote the *true positive* fraction, that is the fraction of cases where the classifier prediction was positive, and the person actually had the disease. Let *f_pos* denote the *false positive* fraction, that is the fraction of cases where the prediction was positive, but the person did not have the disease. Let *t_neg* denote *true negative* and *f_neg* denote *false negative* fractions, which are defined similarly, but for the cases where the classifier prediction was negative.

a. Compute the following metrics for each classifier:
   i. *Accuracy*, defined as $(t_pos + t_neg)/(pos+neg)$, that is, the fraction of the time when the classifier gives the correct classification.
   ii. *Recall* (also known as *sensitivity*) defined as t_pos/pos, that is, how many of the actual positive cases are classified as positive.
   iii. *Precision*, defined as $t_pos/(t_pos+f_pos)$, that is, how often the positive prediction is correct.
   iv. *Specificity*, defined as t_neg/neg.

b. If you intend to use the results of classification to perform further screening for the disease, how would you choose between the classifiers?

c. On the other hand, if you intend to use the result of classification to start medication, where the medication could have harmful effects if given to someone who does not have the disease, how would you choose between the classifiers?

Exercises

11.7 Why is column-oriented storage potentially advantageous in a database system that supports a data warehouse?

11.8 Consider each of the *takes* and *teaches* relations as a fact table; they do not have an explicit measure attribute, but assume each table has a measure attribute *reg_count* whose value is always 1. What would the dimension attributes and dimension tables be in each case. Would the resultant schemas be star schemas or snowflake schemas?

11.9 Consider the star schema from Figure 11.2. Suppose an analyst finds that monthly total sales (sum of the *price* values of all *sales* tuples) have decreased, instead of growing, from April 2018 to May 2018. The analyst wishes to check if there are specific item categories, stores, or customer countries that are responsible for the decrease.

Telegram: @uni_k

www.konkur.in

a. What are the aggregates that the analyst would start with, and what are the relevant drill-down operations that the analyst would need to execute?

b. Write an SQL query that shows the item categories that were responsible for the decrease in sales, ordered by the impact of the category on the sales decrease, with categories that had the highest impact sorted first.

11.10 Suppose half of all the transactions in a clothes shop purchase jeans, and one-third of all transactions in the shop purchase T-shirts. Suppose also that half of the transactions that purchase jeans also purchase T-shirts. Write down all the (nontrivial) association rules you can deduce from the above information, giving support and confidence of each rule.

11.11 The organization of parts, chapters, sections, and subsections in a book is related to clustering. Explain why, and to what form of clustering.

11.12 Suggest how predictive mining techniques can be used by a sports team, using your favorite sport as an example.

Tools

Data warehouse systems are available from Teradata, Teradata Aster, SAP IQ (formerly known as Sybase IQ), and Amazon Redshift, all of which support parallel processing across a large number of machines. A number of databases including Oracle, SAP HANA, Microsoft SQL Server, and IBM DB2 support data warehouse applications by adding features such as columnar storage. There are a number of commercial ETL tools including tools from Informatica, Business Objects, IBM InfoSphere, Microsoft Azure Data Factory, Microsoft SQL Server Integration Services, Oracle Warehouse Builder, and Pentaho Data Integration. Open-source ETL tools include Apache NiFi (nifi.apache.org), Jasper ETL (www.jaspersoft.com/data-integration) and Talend (sourceforge.net/projects/talend-studio). Apache Kafka (kafka.apache.org) can also be used to build ETL systems.

Most database vendors provide OLAP tools as part of their database systems, or as add-on applications. These include OLAP tools from Microsoft Corp., Oracle, IBM and SAP. The Mondrian OLAP server (github.com/pentaho/mondrian) is an open-source OLAP server. Apache Kylin (kylin.apache.org) is an open-source distributed analytics engine which can process data stored in Hadoop, build OLAP cubes and store them in the HBase key-value store, and then query the stored cubes using SQL. Many companies also provide analysis tools for specific applications, such as customer relationship management.

Tools for visualization include Tableau (www.tableau.com), FusionCharts (www.fusioncharts.com), plotly (plot.ly), Datawrapper (www.datawrapper.de), and Google Charts (developers.google.com/chart).

Telegram: @uni_k

www.konkur.in

The Python language is very popular for machine-learning tasks, due to the availability of a number of open-source libraries for machine-learning tasks. The R language is also used widely for statistical analysis and machine learning, for the same reasons. Popular utility libraries in Python include are NumPy (www.numpy.org) which provides operations on arrays and matrices, SciPy (www.scipy.org), which provides linear algebra, optimization and statistics functions, and Pandas (pandas.pydata.org), which provides a relational abstraction of data. Popular machine-learning libraries in Python include SciKit-Learn (scikit-learn.org), which adds image-processing and machine-learning functionality to SciPy. Deep learning libraries in Python include Keras (keras.io), and TensorFlow (www.tensorflow.org) which was developed by Google; TensorFlow provides APIs in several languages, with particularly good support for Python. Text mining is supported by natural language processing libraries, such as NLTK (www.nltk.org) and web crawling libraries, such as Scrapy (scrapy.org). Visualization is supported by libraries such as Matplotlib (matplotlib.org), Plotly (plot.ly) and Bokeh (bokeh.pydata.org).

Open-source tools for data mining include RapidMiner (rapidminer.com), Weka (www.cs.waikato.ac.nz/ml/weka), and Orange (orange.biolab.si). Commercial tools include SAS Enterprise Miner, IBM Intelligent Miner, and Oracle Data Mining.

Further Reading

[Kimball et al. (2008)] and [Kimball and Ross (2013)] provide textbook coverage of data warehouses and multidimensional modeling.

[Mitchell (1997)] is a classic textbook on machine learning and covers classification techniques in detail. [Goodfellow et al. (2016)] is a definitive text on deep learning. [Witten et al. (2011)] and [Han et al. (2011)] provide textbook coverage of data mining. [Agrawal et al. (1993)] introduced the notion of association rules.

Information about the R language and environment may be found at www.r-project.org; information about the SparkR package, which provides an R frontend to Apache Spark, may be found at spark.apache.org/docs/latest/sparkr.html.

[Chakrabarti (2002)], [Manning et al. (2008)] and [Baeza-Yates and Ribeiro-Neto (2011)] provide textbook description of information retrieval, including extensive coverage of data-mining tasks related to textual and hypertext data, such as classification and clustering.

Bibliography

[Agrawal et al. (1993)] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), pages 207–216.

[Baeza-Yates and Ribeiro-Neto (2011)] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, 2nd edition, ACM Press (2011).

Telegram: @uni_k

www.konkur.in

[Chakrabarti (2002)] S. Chakrabarti, *Mining the Web: Discovering Knowledge from HyperText Data*, Morgan Kaufmann (2002).

[Goodfellow et al. (2016)] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press (2016).

[Han et al. (2011)] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd edition, Morgan Kaufmann (2011).

[Kimball and Ross (2013)] R. Kimball and M. Ross, "The Data Warehouse Tookit: The Definitive Guide to Dimensional Modeling", John Wiley and Sons (2013).

[Kimball et al. (2008)] R. Kimball, M. Ross, W. Thornthwaite, J. Mundy, and B. Becker, "The Data Warehouse Lifecycle Toolkit", John Wiley and Sons (2008).

[Manning et al. (2008)] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press (2008).

[Mitchell (1997)] T. M. Mitchell, *Machine Learning*, McGraw Hill (1997).

[Witten et al. (2011)] I. H. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, 3rd edition, Morgan Kaufmann (2011).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in



PART 5

STORAGE MANAGEMENT AND INDEXING

Although a database system provides a high-level view of data, ultimately data have to be stored as bits on one or more storage devices. A vast majority of database systems today store data on magnetic disk, with data having higher performance requirements stored on flash-based solid-state drives. Database systems fetch data into main memory for processing, and write data back to storage for persistence. Data are also copied to tapes and other backup devices for archival storage. The physical characteristics of storage devices play a major role in the way data are stored, in particular because access to a random piece of data on magnetic disk is much slower than main-memory access. Magnetic disk access takes tens of milliseconds, flash-based storage access takes 20 to 100 microseconds, whereas main-memory access takes a tenth of a microsecond.

Chapter 12 begins with an overview of physical storage media, including magnetic disks and flash-based solid-state drives (SSD). The chapter then covers mechanisms to minimize the chance of data loss due to device failures, including RAID. The chapter concludes with a discussion of techniques for efficient disk-block access.

Chapter 13 describes how records are mapped to files, which in turn are mapped to bits on the disk. The chapter then covers techniques for the efficient management of the main-memory buffer for disk-based data. Column-oriented storage, used in data analytics systems, is also covered in this chapter.

Many queries reference only a small proportion of the records in a file. An index is a structure that helps locate desired records of a relation quickly, without examining all records. Chapter 14 describes several types of indices used in database systems.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 12



Physical Storage Systems

In preceding chapters, we have emphasized the higher-level models of a database. For example, at the *conceptual* or *logical* level, we viewed the database, in the relational model, as a collection of tables. Indeed, the logical model of the database is the correct level for database *users* to focus on. This is because the goal of a database system is to simplify and facilitate access to data; users of the system should not be burdened unnecessarily with the physical details of the implementation of the system.

In this chapter, however, as well as in Chapter 13, Chapter 14, Chapter 15, and Chapter 16, we probe below the higher levels as we describe various methods for implementing the data models and languages presented in preceding chapters. We start with characteristics of the underlying storage media, with a particular focus on magnetic disks and flash-based solid-state disks, and then discuss how to create highly reliable storage structures by using multiple storage devices.

12.1 Overview of Physical Storage Media

Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability. Among the media typically available are these:

- **Cache**. The cache is the fastest and most costly form of storage. Cache memory is relatively small; its use is managed by the computer system hardware. We shall not be concerned about managing cache storage in the database system. It is, however, worth noting that database implementors do pay attention to cache effects when designing query processing data structures and algorithms, and we shall return to this issue in later chapters.
- **Main memory**. The storage medium used for data that are available to be operated on is main memory. The general-purpose machine instructions operate on main memory. Main memory may contain tens of gigabytes of data on a personal com-

Telegram: @uni_k

www.konkur.in

puter, and even hundreds to thousands of gigabytes of data in large server systems. It is generally too small (or too expensive) for storing the entire database for very large databases, but many enterprise databases can fit in main memory. However, the contents of main memory are lost in the event of a power failure or system crash; main memory is therefore said to be **volatile**.

- **Flash memory**. Flash memory differs from main memory in that stored data are retained even if power is turned off (or fails)—that is, it is **non-volatile**. Flash memory has a lower cost per byte than main memory, but a higher cost per byte than magnetic disks.

  Flash memory is widely used for data storage in devices such as cameras and cell phones. Flash memory is also used for storing data in "USB flash drives," also known as "pen drives," which can be plugged into the *Universal Serial Bus* (USB) slots of computing devices.

  Flash memory is also increasingly used as a replacement for magnetic disks in personal computers as well as in servers. A **solid-state drive** (**SSD**) uses flash memory internally to store data but provides an interface similar to a magnetic disk, allowing data to be stored or retrieved in units of a block; such an interface is called a *block-oriented interface*. Block sizes typically range from 512 bytes to 8-kilobytes. As of 2018, a 1-terabyte SSD costs around $250. We provide more details about flash memory in Section 12.4.

- **Magnetic-disk storage**. The primary medium for the long-term online storage of data is the magnetic disk drive, which is also referred to as the **hard disk drive** (**HDD**). Magnetic disk, like flash memory, is non-volatile: that is, magnetic disk storage survives power failures and system crashes. Disks may sometimes fail and destroy data, but such failures are quite rare compared to system crashes or power failures.

  To access data stored on magnetic disk, the system must first move the data from disk to main memory, from where they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.

  Disk capacities have grown steadily over the years. As of 2018, the size of magnetic disks ranges from 500 gigabytes to 14 terabytes, and a 1-terabyte disk costs about $50, while an 8-terabyte disk around $150. Although significantly cheaper than SSDs, magnetic disks provide lower performance in terms of number of data access operations that they can support per second. We provide further details about magnetic disks in Section 12.3.

- **Optical storage**. The *digital video disk* (DVD) is an optical storage medium, with data written and read back using a laser light source. The *Blu-ray DVD* format has a capacity of 27 gigabytes to 128 gigabytes, depending on the number of layers supported. Although the original (and still main) use of DVDs was to store video data, they are capable of storing any type of digital data, including backups of

Telegram: @uni_k

www.konkur.in

database contents. DVDs are not suitable for storing active database data since the time required to access a given piece of data can be quite long compared to the time taken by a magnetic disk.

Some DVD versions are read-only, written at the factory where they are produced, other versions support *write-once*, allowing them to be written once, but not overwritten, and some versions can be rewritten multiple times. Disks that can be written only once are called **write-once, read-many** (WORM) disks.

Optical disk **jukebox** systems contain a few drives and numerous disks that can be loaded into one of the drives automatically (by a robot arm) on demand.

- **Tape storage**. Tape storage is used primarily for backup and archival data. *Archival* data refers to data that must be stored safely for a long period of time, often for legal reasons. Magnetic tape is cheaper than disks and can safely store data for many years. However, access to data is much slower because the tape must be accessed sequentially from the beginning of the tape; tapes can be very long, requiring tens to hundreds of seconds to access data. For this reason, tape storage is referred to as **sequential-access** storage. In contrast, magnetic disk and SSD storage are referred to as **direct-access** storage because it is possible to read data from any location on disk.

  Tapes have a high capacity (1 to 12 terabyte capacities are currently available), and can be removed from the tape drive. Tape drives tend to be expensive, but individual tapes are usually significantly cheaper than magnetic disks of the same capacity. As a result, tapes are well suited to cheap archival storage and to transferring large amounts of data between different locations. Archival storage of large video files, as well as storage of large volumes of scientific data, which can range up to many petabytes (1 petabyte = 10^{15} bytes) of data, are two common use cases for tapes.

  Tape libraries (jukeboxes) are used to hold large collections of tapes, allowing automated storage and retrieval of tapes without human intervention.

The various storage media can be organized in a hierarchy (Figure 12.1) according to their speed and their cost. The higher levels are expensive, but fast. As we move down the hierarchy, the cost per bit decreases, whereas the access time increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory.

The fastest storage media—for example, cache and main memory—are referred to as **primary storage**. The media in the next level in the hierarchy—for example, flash memory and magnetic disks—are referred to as **secondary storage**, or **online storage**. The media in the lowest level in the hierarchy—for example, magnetic tape and optical-disk jukeboxes—are referred to as **tertiary storage**, or **offline storage**.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. In the hierarchy shown in Figure 12.1, the storage systems

Telegram: @uni_k

www.konkur.in

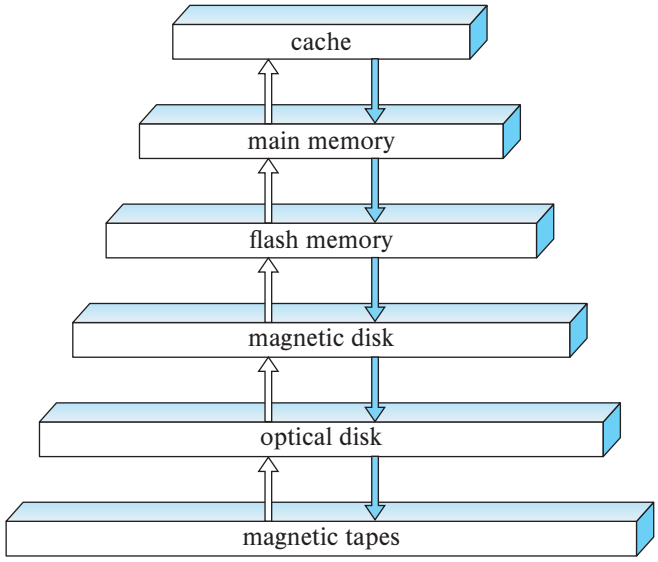


Figure 12.1 Storage device hierarchy.

from main memory up are volatile, whereas the storage systems from flash memory down are non-volatile. Data must be written to non-volatile storage for safekeeping. We shall return to the subject of safe storage of data in the face of system failures later, in Chapter 19.

12.2 Storage Interfaces

Magnetic disks as well as flash-based solid-state disks are connected to a computer system through a high-speed interconnection. Disks typically support either the **Serial ATA** (**SATA**) interface, or the **Serial Attached SCSI** (**SAS**) interface; the SAS interface is typically used only in servers. The SATA-3 version of SATA nominally supports 6 gigabytes per second, allowing data transfer speeds of up to 600 megabytes per second, while SAS version 3 supports data transfer rates of 12 gigabits per second. The **Non-Volatile Memory Express** (**NVMe**) interface is a logical interface standard developed to better support SSDs and is typically used with the PCIe interface (the PCIe interface provides high-speed data transfer internal to computer systems).

While disks are usually connected directly by cables to the disk interface of the computer system, they can be situated remotely and connected by a high-speed network to the computer. In the **storage area network** (**SAN**) architecture, large numbers of disks are connected by a high-speed network to a number of server computers. The disks are usually organized locally using a storage organization technique called *redundant arrays of independent disks* (RAID) (described later, in Section 12.5), to give the servers a logical view of a very large and very reliable disk. Interconnection technologies used

Telegram: @uni_k

www.konkur.in

in storage area networks include iSCSI, which allows SCSI commands to be sent over an IP network, Fiber Channel FC, which supports transfer rates of 1.6 to 12 gigabytes per second, depending on the version, and InfiniBand, which provides very low latency high-bandwidth network communication.

Network attached storage (**NAS**) is an alternative to SAN. NAS is much like SAN, except that instead of the networked storage appearing to be a large disk, it provides a file system interface using networked file system protocols such as NFS or CIFS. Recent years have also seen the growth of **cloud storage**, where data are stored in the cloud and accessed via an API. Cloud storage has a very high latency of tens to hundreds of milliseconds, if the data are not co-located with the database, and is thus not ideal as the underlying storage for databases. However, applications often use cloud storage for storing objects. Cloud-based storage systems are discussed further in Section 21.7.

12.3 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Magnetic disk capacities have been growing steadily year after year, but the storage requirements of large applications have also been growing very fast, in some cases even faster than the growth rate of disk capacities. Very large databases at "web-scale" require thousands to tens of thousands of disks to store their data.[1]

In recent years, SSD storage sizes have grown rapidly, and the cost of SSDs has come down significantly; the increasing affordability of SSDs coupled with their much better performance has resulted in SSDs increasingly becoming a competitor to magnetic disk storage for several applications. However, the fact that the per-byte cost of storage on SSDs is around six to eight times the per-byte cost of storage on magnetic disks means that magnetic disks continue to be the preferred choice for storing very large volumes of data in many applications. Example of such data include video and image data, as well as data that is accessed less frequently, such as user-generated data in many web-scale applications. SSDs have however, increasingly become the preferred choice for enterprise data.

12.3.1 Physical Characteristics of Disks

Figure 12.2 shows a schematic diagram of a magnetic disk, while Figure 12.3 shows the internals of an actual magnetic disk. Each disk **platter** has a flat, circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass.

When the disk is in use, a drive motor spins it at a constant high speed, typically 5400 to 10,000 revolutions per minute, depending on the model. There is a read-write head positioned just above the surface of the platter. The disk surface is logically di-

[1] We study later, in Chapter 21, how to partition such large amounts of data across multiple nodes in a parallel computing system.

Telegram: @uni_k

www.konkur.in

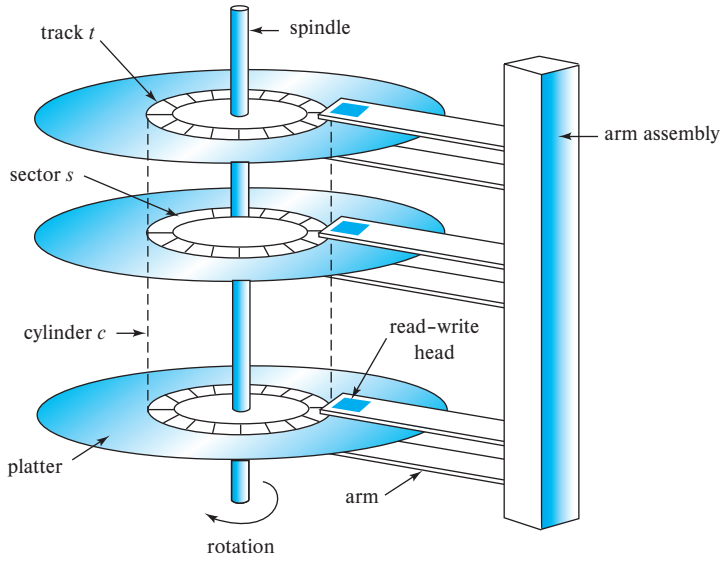


Figure 12.2 Schematic diagram of a magnetic disk.

vided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk. Sector sizes are typically 512 bytes, and current generation disks have between 2 billion and 24 billion sectors. The inner tracks (closer to the spindle) are of smaller length than the outer tracks, and the outer tracks contain more sectors than the inner tracks.

The **read-write head** stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material.

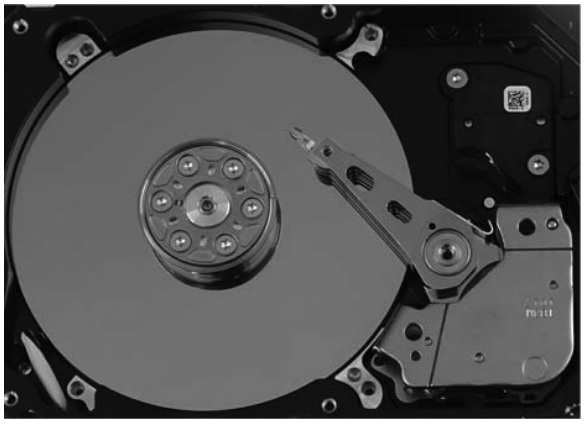


Figure 12.3 Internals of an actual magnetic disk.

Telegram: @uni_k

www.konkur.in

Each side of a platter of a disk has a read-write head that moves across the platter to access different tracks. A disk typically contains many platters, and the read-write heads of all the tracks are mounted on a single assembly called a **disk arm** and move together. The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as **head-disk assemblies**. Since the heads on all the platters move together, when the head on one platter is on the *i*th track, the heads on all other platters are also on the *i*th track of their respective platters. Hence, the *i*th tracks of all the platters together are called the *i*th **cylinder**.

The read-write heads are kept as close as possible to the disk surface to increase the recording density. The head typically floats or flies only microns from the disk surface; the spinning of the disk creates a small breeze, and the head assembly is shaped so that the breeze keeps the head floating just above the disk surface. Because the head floats so close to the surface, platters must be machined carefully to be flat.

Head crashes can be a problem. If the head contacts the disk surface, the head can scrape the recording medium off the disk, destroying the data that had been there. In older-generation disks, the head touching the surface caused the removed medium to become airborne and to come between the other heads and their platters, causing more crashes; a head crash could thus result in failure of the entire disk. Current-generation disk drives use a thin film of magnetic metal as recording medium. They are much less susceptible to failure of the entire disk, but are susceptible to failure of individual sectors.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive; in modern disk systems, the disk controller is implemented within the disk drive unit. A disk controller accepts high-level commands to read or write a sector, and initiates actions, such as moving the disk arm to the right track and actually reading or writing the data. Disk controllers also attach **checksums** to each sector that is written; the checksum is computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum; if the data are corrupted, with a high probability the newly computed checksum will not match the stored checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure.

Another interesting task that disk controllers perform is **remapping of bad sectors**. If the controller detects that a sector is damaged when the disk is initially formatted, or when an attempt is made to write the sector, it can logically map the sector to a different physical location (allocated from a pool of extra sectors set aside for this purpose). The remapping is noted on disk or in non-volatile memory, and the write is carried out on the new location.

12.3.2 Performance Measures of Disks

The main measures of the qualities of a disk are capacity, access time, data-transfer rate, and reliability.

Telegram: @uni_k

www.konkur.in

Access time is the time from when a read or write request is issued to when data transfer begins. To access (i.e., to read or write) data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called the **seek time**, and it increases with the distance that the arm must move. Typical seek times range from 2 to 20 milliseconds depending on how far the track is from the initial arm position. Smaller disks tend to have lower seek times since the head has to travel a smaller distance.

The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests. If all tracks have the same number of sectors, and we disregard the time required for the head to start moving and to stop moving, we can show that the average seek time is one-third the worst-case seek time. Taking these factors into account, the average seek time is around one-half of the maximum seek time. Average seek times currently range between 4 and 10 milliseconds, depending on the disk model.[2]

Once the head has reached the desired track, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**. Rotational speeds of disks today range from 5400 rotations per minute (90 rotations per second) up to 15,000 rotations per minute (250 rotations per second), or, equivalently, 4 milliseconds to 11.1 milliseconds per rotation. On an average, one-half of a rotation of the disk is required for the beginning of the desired sector to appear under the head. Thus, the **average latency time** of the disk is one-half the time for a full rotation of the disk. Disks with higher rotational speeds are used for applications where latency needs to be minimized.

The access time is then the sum of the seek time and the latency; average access times range from 5 to 20 milliseconds depending on the disk model. Once the first sector of the data to be accessed has come under the head, data transfer begins. The **data-transfer rate** is the rate at which data can be retrieved from or stored to the disk. Current disk systems support maximum transfer rates of 50 to 200 megabytes per second; transfer rates are significantly lower than the maximum transfer rates for inner tracks of the disk, since they have fewer sectors. For example, a disk with a maximum transfer rate of 100 megabytes per second may have a sustained transfer rate of around 30 megabytes per second on its inner tracks.

Requests for disk I/O are typically generated by the file system but can be generated directly by the database system. Each request specifies the address on the disk to be referenced; that address is in the form of a *block number*. A **disk block** is a logical unit of storage allocation and retrieval, and block sizes today typically range from 4 to 16

[2]Smaller 2.5-inch diameter disks have a lesser arm movement distance than larger 3.5-inch disks, and thus have lower seek times. As a result 2.5-inch disks have been the preferred choice for applications where latency needs to be minimized, although SSDs are increasingly preferred for such applications. Larger 3.5-inch diameter disks have a lower cost per byte and are used in data storage applications where cost is an important factor.

Telegram: @uni_k

www.konkur.in

kilobytes. Data are transferred between disk and main memory in units of blocks. The term **page** is often used to refer to blocks, although in a few contexts (such as flash memory) they refer to different things.

A sequence of requests for blocks from disk may be classified as a sequential access pattern or a random access pattern. In a **sequential access** pattern, successive requests are for successive block numbers, which are on the same track, or on adjacent tracks. To read blocks in sequential access, a disk seek may be required for the first block, but successive requests would either not require a seek, or require a seek to an adjacent track, which is faster than a seek to a track that is farther away. Data transfer rates are highest with a sequential access pattern, since seek time is minimal.

In contrast, in a **random access** pattern, successive requests are for blocks that are randomly located on disk. Each such request would require a seek. The number of **I/O operations per second** (**IOPS**), that is, the number random block accesses that can be satisfied by a disk in a second, depends on the access time, and the block size, and the data transfer rate of the disk. With a 4-kilobyte block size, current generation disks support between 50 and 200 IOPS, depending on the model. Since only a small amount (one block) of data are read per seek, the data transfer rate is significantly lower with a random access pattern than with a sequential access pattern.

The final commonly used measure of a disk is the **mean time to failure** (**MTTF**),[3] which is a measure of the reliability of the disk. The mean time to failure of a disk (or of any other system) is the amount of time that, on average, we can expect the system to run continuously without any failure. According to vendors' claims, the mean time to failure of disks today ranges from 500,000 to 1,200,000 hours—about 57 to 136 years. In practice the claimed mean time to failure is computed on the probability of failure when the disk is new—the figure means that given 1000 relatively new disks, if the MTTF is 1,200,000 hours, on an average one of them will fail in 1200 hours. A mean time to failure of 1,200,000 hours does not imply that the disk can be expected to function for 136 years! Most disks have an expected life span of about 5 years and have significantly higher rates of failure once they become more than a few years old.

12.4 Flash Memory

There are two types of flash memory, NOR flash and NAND flash. NAND flash is the variant that is predominantly used for data storage. Reading from NAND flash requires an entire *page* of data, which is very commonly 4096 bytes, to be fetched from NAND flash into main memory. Pages in a NAND flash are thus similar to sectors in a magnetic disk.

[3]The term **mean time between failures** (**MTBF**) is often used to refer to MTTF in the context of disk drives, although technically MTBF should only be used in the context of systems that can be repaired after failure, and may fail again; MTBF would then be the sum of MTTF and the mean time to repair. Magnetic disks can almost never be repaired after a failure.

Telegram: @uni_k

www.konkur.in

Solid-state disks (SSDs) are built using NAND flash and provide the same block-oriented interface as disk storage. Compared to magnetic disks, SSDs can provide much faster random access: the latency to retrieve a page of data ranges from 20 to 100 microseconds for SSDs, whereas a random access on disk would take 5 to 10 milliseconds. The data transfer rate of SSDs is higher than that of magnetic disks and is usually limited by the interconnect technology; transfer rates range from around 500 megabytes per second with SATA interfaces, up to 3 gigabytes per second using NVMe PCIe interfaces, depending on the specific SSD model, in contrast to a maximum of about 200 megabytes per second with magnetic disk. The power consumption of SSDs is also significantly lower than that of magnetic disks.

Writes to flash memory are a little more complicated. A write to a page of flash memory typically takes about 100 microseconds. However, once written, a page of flash memory cannot be directly overwritten. Instead, it has to be erased and rewritten subsequently. The erase operation must be performed on a group of pages, called an **erase block**, erasing all the pages in the block, and takes about 2 to 5 milliseconds. An erase block (often referred to as just "block" in flash literature), is typically 256 kilobytes to 1 megabyte, and contains around 128 to 256 pages. Further, there is a limit to how many times a flash page can be erased, typically around 100,000 to 1,000,000 times. Once this limit is reached, errors in storing bits are likely to occur.

Flash memory systems limit the impact of both the slow erase speed and the update limits by mapping logical page numbers to physical page numbers. When a logical page is updated, it can be remapped to any already erased physical page, and the original location can be erased later. Each physical page has a small area of memory where its logical address is stored; if the logical address is remapped to a different physical page, the original physical page is marked as deleted. Thus, by scanning the physical pages, we can find where each logical page resides. The logical-to-physical page mapping is replicated in an in-memory **translation table** for quick access.

Blocks containing multiple deleted pages are periodically erased, taking care to first copy nondeleted pages in those blocks to a different block (the translation table is updated for these nondeleted pages). Since each physical page can be updated only a fixed number of times, physical pages that have been erased many times are assigned "cold data," that is, data that are rarely updated, while pages that have not been erased many times are used to store "hot data," that is, data that are updated frequently. This principle of evenly distributing erase operations across physical blocks is called **wear leveling** and is usually performed transparently by flash-memory controllers. If a physical page is damaged due to an excessive number of updates, it can be removed from usage, without affecting the flash memory as a whole.

All the above actions are carried out by a layer of software called the **flash translation layer**; above this layer, flash storage looks identical to magnetic disk storage, providing the same page/sector-oriented interface, except that flash storage is much faster. File systems and database storage structures can thus see an identical logical view of the underlying storage structure, regardless of whether it is flash or magnetic storage.

Telegram: @uni_k

www.konkur.in



SSD performance is usually expressed in terms of:

1. The *number of random block reads per second*, with 4-kilobyte blocks being the standard. Typical values in 2018 are about 10,000 random reads per second (also referred to as 10,000 IOPS) with 4-kilobyte blocks, although some models support higher rates.

   Unlike magnetic disks, SSDs can support multiple random requests in parallel, with 32 parallel requests being commonly supported; a flash disk with SATA interface supports nearly 100,000 random 4-kilobyte block reads in a second with 32 requests sent in parallel, while SSDs connected using NVMe PCIe can support over 350,000 random 4-kilobyte block reads per second. These numbers are specified as QD-1 for rates without parallelism and QD-n for n-way parallelism, with QD-32 being the most commonly used number.

2. The *data transfer rate* for sequential reads and sequential writes. Typical rates for both sequential reads and sequential writes are 400 to 500 megabytes per second for SSDs with a SATA 3 interface, and 2 to 3 gigabytes per second for SSDs using NVMe over the PCIe 3.0x4 interface.

3. The *number of random block writes per second*, with 4-kilobyte blocks being the standard. Typical values in 2018 are about 40,000 random 4-kilobyte writes per second for QD-1 (without parallelism), and around 100,000 IOPS for QD-32. although some models support higher rates for both QD-1 and QD-32.

Hybrid disk drives are hard-disk systems that combine magnetic storage with a smaller amount of flash memory, which is used as a cache for frequently accessed data. Frequently accessed data that are rarely updated are ideal for caching in flash memory.

Telegram: @uni_k

www.konkur.in

Modern SAN and NAS systems support the use of a combination of magnetic disks and SSDs, and they can be configured to use the SSDs as a cache for data that reside on magnetic disks.

12.5 RAID

The data-storage requirements of some applications (in particular web, database, and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.

Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.

A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (**RAID**), have been proposed to achieve improved performance and reliability.

In the past, system designers viewed storage systems composed of several small, cheap disks as a cost-effective alternative to using large, expensive disks; the cost per megabyte of the smaller disks was less than that of larger disks. In fact, the I in RAID, which now stands for *independent*, originally stood for *inexpensive*. Today, however, all disks are physically small, and larger-capacity disks actually have a lower cost per megabyte. RAID systems are used for their higher reliability and higher performance rate, rather than for economic reasons. Another key justification for RAID use is easier management and operations.

12.5.1 Improvement of Reliability via Redundancy

Let us first consider reliability. The chance that at least one disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the mean time to failure of a disk is 100,000 hours, or slightly over 11 years. Then, the mean time to failure of some disk in an array of 100 disks will be 100,000/100 = 1000 hours, or around 42 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data (as discussed in Section 12.3.1). Such a high frequency of data loss is unacceptable.

The solution to the problem of reliability is to introduce **redundancy**; that is, we store extra information that is not needed normally but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost, so the effective mean time to failure is increased, provided that we count only failures that lead to loss of data or to non-availability of data.

Telegram: @uni_k

www.konkur.in

The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, *shadowing*). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is repaired.

The mean time to failure (where failure is the loss of data) of a mirrored disk depends on the mean time to failure of the individual disks, as well as on the **mean time to repair**, which is the time it takes (on an average) to replace a failed disk and to restore the data on it. Suppose that the failures of the two disks are *independent*; that is, there is no connection between the failure of one disk and the failure of the other. Then, if the mean time to failure of a single disk is 100,000 hours, and the mean time to repair is 10 hours, the **mean time to data loss** of a mirrored disk system is $100,000^2/(2 * 10) = 500 * 10^6$ hours, or 57,000 years! (We do not go into the derivations here; references in the bibliographical notes provide the details.)

You should be aware that the assumption of independence of disk failures is not valid. Power failures and natural disasters such as earthquakes, fires, and floods may result in damage to both disks at the same time. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single-disk systems. Mirrored-disk systems with mean time to data loss of about 500,000 to 1,000,000 hours, or 55 to 110 years, are available today.

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. Power failures are not a concern if there is no data transfer to disk in progress when they occur. However, even with mirroring of disks, if writes are in progress to the same block in both disks, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. The solution to this problem is to write one copy first, then the next, so that one of the two copies is always consistent. Some extra actions are required when we restart after a power failure, to recover from incomplete writes. This matter is examined in Practice Exercise 12.6.

12.5.2 Improvement in Performance via Parallelism

Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

With multiple disks, we can improve the transfer rate as well (or instead) by **striping data** across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**. For example, if we have an array of eight disks, we write bit i of each byte to disk i. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a single disk, but each access can read eight times as much data

Telegram: @uni_k

www.konkur.in

Block-level striping stripes blocks across multiple disks. It treats the array of disks as a single large disk, and it gives blocks logical numbers; we assume the block numbers start from 0. With an array of n disks, block-level striping assigns logical block i of the disk array to disk $(i \bmod n) + 1$; it uses the $\lfloor i/n \rfloor$th physical block of the disk to store logical block i. For example, with eight disks, logical block 0 is stored in physical block 0 of disk 1, while logical block 11 is stored in physical block 1 of disk 4. When reading a large file, block-level striping fetches n blocks at a time in parallel from the n disks, giving a high data-transfer rate for large reads. When a single block is read, the data-transfer rate is the same as on one disk, but the remaining $n-1$ disks are free to perform other actions.

Block-level striping offers several advantages over bit-level striping, including the ability to support a larger number of block reads per second, and lower latency for single block reads. As a result, bit-level striping is not used in any practical system.

In summary, there are two main goals of parallelism in a disk system:

1. Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
2. Parallelize large accesses so that the response time of large accesses is reduced.

12.5.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability. Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with "parity blocks".

Blocks in a RAID system are partitioned into sets, as we shall see. For a given set of blocks, a **parity block** can be computed and stored on disk; the ith bit of the parity block is computed as the "exclusive or" (XOR) of the ith bits of the all blocks in the set. If the contents of any one of the blocks in a set is lost due to a failure, the block contents can be recovered by computing the bitwise-XOR of the remaining blocks in the set, along with the parity block.

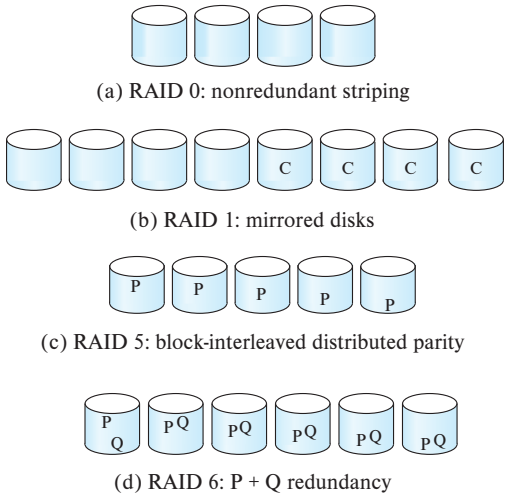
Whenever a block is written, the parity block for its set must be recomputed and written to disk. The new value of the parity block can be computed by either (i) reading all the other blocks in the set from disk and computing the new parity block, or (ii) by computing the XOR of the old value of the parity block with the old and new value of the updated block.

These schemes have different cost-performance trade-offs. The schemes are classified into **RAID levels.**[4]. Figure 12.4 illustrates the four levels that are used in practice. In the figure, P indicates error-correcting bits, and C indicates a second copy of the data. For all levels, the figure depicts four disks' worth of data, and the extra disks depicted are used to store redundant information for failure recovery.

[4]There are 7 different RAID levels, numbered 0 to 6; Levels 2, 3, and 4 are not used in practice anymore and thus are not covered in the text

Telegram: @uni_k

www.konkur.in



(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks

(c) RAID 5: block-interleaved distributed parity

(d) RAID 6: P + Q redundancy

Figure 12.4 RAID levels.

- **RAID level 0** refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits). Figure 12.4a shows an array of size 4.

- **RAID level 1** refers to disk mirroring with block striping. Figure 12.4b shows a mirrored organization that holds four disks' worth of data.

  Note that some vendors use the term **RAID level 1+0** or **RAID level 10** to refer to mirroring with striping, and they use the term RAID level 1 to refer to mirroring without striping. Mirroring without striping can also be used with arrays of disks, to give the appearance of a single large, reliable disk: if each disk has M blocks, logical blocks 0 to $M - 1$ are stored on disk 0, M to $2M - 1$ on disk 1(the second disk), and so on, and each disk is mirrored.[5]

- **RAID level 5** refers to block-interleaved distributed parity. The data and parity are partitioned among all $N + 1$ disks. For each set of N logical blocks, one of the disks stores the parity, and the other N disks store the blocks. The parity blocks are stored on different disks for different sets of N blocks. Thus, all disks can participate in satisfying read requests.[6]

  Figure 12.4c shows the setup. The P's are distributed across all the disks. For example, with an array of five disks, the parity block, labeled Pk, for logical blocks

[5]Note that some vendors use the term RAID 0+1 to refer to a version of RAID that uses striping to create a RAID 0 array, and mirrors the array onto another array, with the difference from RAID 1 being that if a disk fails, the RAID 0 array containing the disk becomes unusable. The mirrored array can still be used, so there is no loss of data. This arrangement is inferior to RAID 1 when a disk has failed, since the other disks in the RAID 0 array can continue to be used in RAID 1, but remain idle in RAID 0+1.

[6]In RAID level 4 (which is not used in practice) all parity blocks are stored on one disk. That disk would not be useful for reads, and it would also have a higher load than other disks if

Telegram: @uni_k

www.konkur.in

$4k, 4k + 1, 4k + 2, 4k + 3$ is stored in disk k mod 5; the corresponding blocks of the other four disks store the four data blocks $4k$ to $4k + 3$. The following table indicates how the first 20 blocks, numbered 0 to 19, and their parity blocks are laid out. The pattern shown gets repeated on further blocks.

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

Note that a parity block cannot store parity for blocks in the same disk, since then a disk failure would result in loss of data as well as of parity, and hence would not be recoverable.

- **RAID level 6**, the P + Q redundancy scheme, is much like RAID level 5, but it stores extra redundant information to guard against multiple disk failures. Instead of using parity, level 6 uses error-correcting codes such as the Reed-Solomon codes (see the bibliographical notes). In the scheme in Figure 12.4g, two bits of redundant data are stored for every four bits of data—unlike one parity bit in level 5—and the system can tolerate two disk failures.

  The letters P and Q in the figure denote blocks containing the two corresponding error-correcting blocks for a given set of data blocks. The layout of blocks is an extension of that for RAID 5. For example, with six disks, the two parity blocks, labeled Pk and Qk, for logical blocks $4k, 4k + 1, 4k + 2, and 4k + 3$ are stored in disk k mod 6 and $(k + 1)$ mod 6, and the corresponding blocks of the other four disks store the four data blocks $4k$ to $4k + 3$.

Finally, we note that several variations have been proposed to the basic RAID schemes described here, and different vendors use different terminologies for the variants. Some vendors support nested schemes that create multiple separate RAID arrays, and then stripe data across the RAID arrays; one of RAID levels 1, 5 or 6 is chosen for the individual arrays. References to further information on this idea are provided in the Further Reading section at the end of the chapter.

12.5.4 Hardware Issues

RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called **software RAID**. However, there are significant benefits to be had by building special-purpose hardware to support RAID, which we outline below; systems with special hardware support are called **hardware RAID** systems.

Telegram: @uni_k

www.konkur.in

Hardware RAID implementations can use non-volatile RAM to record writes before they are performed. In case of power failure, when the system comes back up, it retrieves information about any incomplete writes from non-volatile RAM and then completes the writes. Normal operations can then commence.

In contrast, with software RAID extra work needs to be done to detect blocks that may have been partially written before power failure. For RAID 1, all blocks of the disks are scanned to see if any pair of blocks on the two disks have different contents. For RAID 5, the disks need to be scanned and parity recomputed for each set of blocks and compared to the stored parity. Such scans take a long time, and they are done in the background using a small fraction of the disks' available bandwidth. See Practice Exercise 12.6 for details of how to recover data to the latest value, when an inconsistency is detected; we revisit this issue in the context of database system recovery in Section 19.2.1. The RAID system is said to be *resynchronizing* (or *resynching*) during this phase; normal reads and writes are allowed while resynchronization is in progress, but a failure of a disk during this phase could result in data loss for blocks with incomplete writes. Hardware RAID does not have this limitation.

Even if all writes are completed properly, there is a small chance of a sector in a disk becoming unreadable at some point, even though it was successfully written earlier. Reasons for loss of data on individual sectors could range from manufacturing defects to data corruption on a track when an adjacent track is written repeatedly. Such loss of data that were successfully written earlier is sometimes referred to as a *latent failure*, or as *bit rot*. When such a failure happens, if it is detected early the data can be recovered from the remaining disks in the RAID organization. However, if such a failure remains undetected, a single disk failure could lead to data loss if a sector in one of the other disks has a latent failure.

To minimize the chance of such data loss, good RAID controllers perform **scrubbing**; that is, during periods when disks are idle, every sector of every disk is read, and if any sector is found to be unreadable, the data are recovered from the remaining disks in the RAID organization, and the sector is written back. (If the physical sector is damaged, the disk controller would remap the logical sector address to a different physical sector on disk.)

Server hardware is often designed to permit **hot swapping**; that is, faulty disks can be removed and replaced by new ones without turning power off. The RAID controller can detect that a disk was replaced by a new one and can immediately proceed to reconstruct the data that was on the old disk, and write it to the new disk. Hot swapping reduces the mean time to repair, since replacement of a disk does not have to wait until a time when the system can be shut down. In fact, many critical systems today run on a 24×7 schedule; that is, they run 24 hours a day, 7 days a week, providing no time for shutting down and replacing a failed disk. Further, many RAID implementations assign a spare disk for each array (or for a set of disk arrays). If a disk fails, the spare disk is immediately used as a replacement. As a result, the mean time to repair is reduced greatly, minimizing the chance of any data loss. The failed disk can be replaced at leisure.

Telegram: @uni_k

www.konkur.in

The power supply, or the disk controller, or even the system interconnection in a RAID system could become a single point of failure that could stop the functioning of the RAID system. To avoid this possibility, good RAID implementations have multiple redundant power supplies (with battery backups so they continue to function even if power fails). Such RAID systems have multiple disk interfaces and multiple interconnections to connect the RAID system to the computer system (or to a network of computer systems). Thus, failure of any single component will not stop the functioning of the RAID system.

12.5.5 Choice of RAID Level

The factors to be taken into account in choosing a RAID level are:

- Monetary cost of extra disk-storage requirements.
- Performance requirements in terms of number of I/O operations per second.
- Performance when a disk has failed.
- Performance during rebuild (i.e., while the data in a failed disk are being rebuilt on a new disk).

The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk. The **rebuild performance** of a RAID system may be an important factor if continuous availability of data is required, as it is in high-performance database systems. Furthermore, since rebuild time can form a significant part of the repair time, rebuild performance also influences the mean time to data loss.

RAID level 0 is used in a few high-performance applications where data safety is not critical, but not anywhere else.

RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance. RAID level 5 has a lower storage overhead than level 1, but it has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice.

Disk-storage capacities have been increasing rapidly for many years. Capacities were effectively doubling every 13 months at one point; although the current rate of growth is much less now, capacities have continued to increase rapidly. The cost per byte of disk storage has been falling at about the same rate as the capacity increase. As a result, for many existing database applications with moderate storage requirements, the monetary cost of the extra disk storage needed for mirroring has become relatively small (the extra monetary cost, however, remains a significant issue for storage-intensive applications such as video data storage). Disk access speeds have not improved significantly in recent years, while the number of I/O operations required per second has increased tremendously, particularly for web application servers.

Telegram: @uni_k

www.konkur.in

RAID level 5 has a significant overhead for random writes, since a single random block write requires 2 block reads (to get the old values of the block and parity block) and 2 block writes to write these blocks back. In contrast, the overhead is low for large sequential writes, since the parity block can be computed from the new blocks in most cases, without any reads. RAID level 1 is therefore the RAID level of choice for many applications with moderate storage requirements and high random I/O requirements.

RAID level 6 offers better reliability than level 1 or 5, since it can tolerate two disk failures without losing data. In terms of performance during normal operation, it is similar to RAID level 5, but it has a higher storage cost than RAID level 5. RAID level 6 is used in applications where data safety is very important. It is being viewed as increasingly important since latent sector failures are not uncommon, and it may take a long time to be detected and repaired. A failure of a different disk before a latent failure is detected and repaired would then be similar to a two-disk failure for that sector and result in loss of data of that sector. RAID levels 1 and 5 would suffer from data loss in such a scenario, unlike level 6.

Mirroring can also be extended to store copies on three disks instead of two to survive two-disk failures. Such triple-redundancy schemes are not commonly used in RAID systems, although they are used in distributed file systems, where data are stored in multiple machines, since the probability of machine failure is significantly higher than that of disk failure.

RAID system designers have to make several other decisions as well. For example, how many disks should there be in an array? How many bits should be protected by each parity bit? If there are more disks in an array, data-transfer rates are higher, but the system will be more expensive. If there are more bits protected by a parity bit, the space overhead due to parity bits is lower, but there is an increased chance that a second disk will fail before the first failed disk is repaired, and that will result in data loss.

12.5.6 Other RAID Applications

The concepts of RAID have been generalized to other storage devices, including in the flash memory devices within SSDs, arrays of tapes, and even to the broadcast of data over wireless systems. Individual flash pages have a higher rate of data loss than sectors of magnetic disks. Flash devices such as SSDs implement RAID internally, to ensure that the device does not lose data due to the loss of a flash page. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data are split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units.

12.6 Disk-Block Access

Requests for disk I/O are generated by the database system, with the query processing subsystem responsible for most of the disk I/O. Each request specifies a disk identifier and a logical block number on the disk; in case

Telegram: @uni_k

www.konkur.in

system files, the request instead specifies the file identifier and a block number within the file. Data are transferred between disk and main memory in units of blocks.

As we saw earlier, a sequence of requests for blocks from disk may be classified as a sequential access pattern or a random access pattern. In a *sequential access* pattern, successive requests are for successive block numbers, which are on the same track, or on adjacent tracks. In contrast, in a *random access* pattern, successive requests are for blocks that are randomly located on disk. Each such request would require a seek, resulting in a longer access time, and a lower number of random I/O operations per second.

A number of techniques have been developed for improving the speed of access to blocks, by minimizing the number of accesses, and in particular minimizing the number of random accesses. We describe these techniques below. Reducing the number of random accesses is very important for data stored on magnetic disks; SSDs support much faster random access than do magnetic disks, so the impact of random access is less with SSDs, but data access from SSDs can still benefit from some of the techniques described below.

- **Buffering**. Blocks that are read from disk are stored temporarily in an in-memory buffer, to satisfy future requests. Buffering is done by both the operating system and the database system. Database buffering is discussed in more detail in Section 13.5.
- **Read-ahead**. When a disk block is accessed, consecutive blocks from the same track are read into an in-memory buffer even if there is no pending request for the blocks. In the case of sequential access, such **read-ahead** ensures that many blocks are already in memory when they are requested, and it minimizes the time wasted in disk seeks and rotational latency per block read. Operating systems also routinely perform read-ahead for consecutive blocks of an operating system file. Read-ahead is, however, not very useful for random block accesses.
- **Scheduling**. If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting the blocks in the order in which they will pass under the heads. If the desired blocks are on different cylinders, it is advantageous to request the blocks in an order that minimizes disk-arm movement. **Disk-arm–scheduling** algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed. A commonly used algorithm is the **elevator algorithm**, which works in the same way many elevators do. Suppose that, initially, the arm is moving from the innermost track toward the outside of the disk. Under the elevator algorithm's control, for each track for which there is an access request, the arm stops at that track, services requests for the track, and then continues moving outward until there are no waiting requests for tracks farther out. At this point, the arm changes direction and moves toward the inside, again stopping at each track for which there is a re-

Telegram: @uni_k

www.konkur.in

quest, until it reaches a track where there is no request for tracks farther toward the center. Then, it reverses direction and starts a new cycle.

Disk controllers usually perform the task of reordering read requests to improve performance, since they are intimately aware of the organization of blocks on disk, of the rotational position of the disk platters, and of the position of the disk arm. To enable such reordering, the disk controller interface must allow multiple requests to be added to a queue; results may be returned in a different order from the request order.

- **File organization**. To reduce block-access time, we can organize blocks on disk in a way that corresponds closely to the way we expect data to be accessed. For example, if we expect a file to be accessed sequentially, then we should ideally keep all the blocks of the file sequentially on adjacent cylinders. Modern disks hide the exact block location from the operating system but use a logical numbering of blocks that gives consecutive numbers to blocks that are adjacent to each other. By allocating consecutive blocks of a file to disk blocks that are consecutively numbered, operating systems ensure that files are stored sequentially.

  Storing a large file in a single long sequence of consecutive blocks poses challenges to disk block allocation; instead, operating systems allocate some number of consecutive blocks (an **extent**) at a time to a file. Different extents allocated to a file may not be adjacent to each other on disk. Sequential access to the file needs one seek per extent, instead of one seek per block if blocks are randomly allocated; with large enough extents, the cost of seeks relative to data transfer costs can be minimized.

  Over time, a sequential file that has multiple small appends may become **fragmented**; that is, its blocks become scattered all over the disk. To reduce fragmentation, the system can make a backup copy of the data on disk and restore the entire disk. The restore operation writes back the blocks of each file contiguously (or nearly so). Some systems (such as different versions of the Windows operating system) have utilities that scan the disk and then move blocks to decrease the fragmentation. The performance increases realized from these techniques can be quite significant.

- **Non-volatile write buffers**. Since the contents of main memory are lost in a power failure, information about database updates has to be recorded on disk to survive possible system crashes. For this reason, the performance of update-intensive database applications, such as transaction-processing systems, is heavily dependent on the latency of disk writes.

  We can use *non-volatile random-access memory* (NVRAM) to speed up disk writes. The contents of NVRAM are not lost in power failure. NVRAM was implemented using battery-backed-up RAM in earlier days, but flash memory is currently the primary medium for non-volatile write buffering. The idea is that, when the database system (or the operating system) requests that a block be written to disk, the disk controller writes the block to a **non-volatile write buffer** and imme-

Telegram: @uni_k

www.konkur.in

diately notifies the operating system that the write completed successfully. The controller can subsequently write the data to their destination on disk in a way that minimizes disk arm movement, using the elevator algorithm, for example. If such write reordering is done without using non-volatile write buffers, the database state may become inconsistent in the event of a system crash; recovery algorithms that we study later in Chapter 19 depend on writes being written in the specified order. When the database system requests a block write, it notices a delay only if the NVRAM buffer is full. On recovery from a system crash, any pending buffered writes in the NVRAM are written back to the disk. NVRAM buffers are found in certain high-end disks, but are more frequently found in RAID controllers.

In addition to the above low-level optimizations, optimizations to minimize random accesses can be done at a higher level, by clever design of query processing algorithms. We study efficient query processing techniques in Chapter 15.

12.7 Summary

- Several types of data storage exist in most computer systems. They are classified by the speed with which they can access data, by their cost per unit of data to buy the memory, and by their reliability. Among the media available are cache, main memory, flash memory, magnetic disks, optical disks, and magnetic tapes.
- Magnetic disks are mechanical devices, and data access requires a read–write head to move to the required cylinder, and the rotation of the platters must then bring the required sector under the read–write head. Magnetic disks thus have a high latency for data access.
- SSDs have a much lower latency for data access, and higher data transfer bandwidth than magnetic disks. However, they also have a higher cost per byte than magnetic disks.
- Disks are vulnerable to failure, which could result in loss of data stored on the disk. We can reduce the likelihood of irretrievable data loss by retaining multiple copies of data.
- Mirroring reduces the probability of data loss greatly. More sophisticated methods based on redundant arrays of independent disks (RAID) offer further benefits. By striping data across disks, these methods offer high throughput rates on large accesses; by introducing redundancy across disks, they improve reliability greatly.
- Several different RAID organizations are possible, each with different cost, performance, and reliability characteristics. RAID level 1 (mirroring) and RAID level 5 are the most commonly used.
- Several techniques have been developed to optimize disk block access, such as read ahead, buffering, disk arm scheduling, prefetching, and non-volatile write buffers.

Telegram: @uni_k

www.konkur.in

Review Terms

- Physical storage media
  - Cache
  - Main memory
  - Flash memory
  - Magnetic disk
  - Optical storage
  - Tape storage
- Volatile storage
- Non-volatile storage
- Sequential-access
- Direct-access
- Storage interfaces
  - Serial ATA (SATA)
  - Serial Attached SCSI (SAS)
  - Non-Volatile Memory Express (NVMe)
  - Storage area network (SAN)
  - Network attached storage (NAS)
- Magnetic disk
  - Platter
  - Hard disks
  - Tracks
  - Sectors
  - Read–write head
  - Disk arm
  - Cylinder
  - Disk controller
  - Checksums
  - Remapping of bad sectors
- Disk block
- Performance measures of disks
  - Access time
  - Seek time
  - Latency time
  - I/O operations per second (IOPS)
  - Rotational latency
  - Data-transfer rate
  - Mean time to failure (MTTF)
- Flash Storage
  - Erase Block
  - Wear leveling
  - Flash translation table
  - Flash Translation Layer
- Storage class memory
  - 3D-XPoint
- Redundant arrays of independent disks (RAID)
  - Mirroring
  - Data striping
  - Bit-level striping
  - Block-level striping
- RAID levels
  - Level 0 (block striping, no redundancy)
  - Level 1 (block striping, mirroring)
  - Level 5 (block striping, distributed parity)

Telegram: @uni_k

www.konkur.in

- Level 6 (block striping, P + Q redundancy)
- Rebuild performance
- Software RAID
- Hardware RAID
- Hot swapping
- Optimization of disk-block access
- Disk-arm scheduling
- Elevator algorithm
- File organization
- Defragmenting
- Non-volatile write buffers
- Log disk

Practice Exercises

12.1 SSDs can be used as a storage layer between memory and magnetic disks, with some parts of the database (e.g., some relations) stored on SSDs and the rest on magnetic disks. Alternatively, SSDs can be used as a buffer or cache for magnetic disks; frequently used blocks would reside on the SSD layer, while infrequently used blocks would reside on magnetic disk.

a. Which of the two alternatives would you choose if you need to support real-time queries that must be answered within a guaranteed short period of time? Explain why.

b. Which of the two alternatives would you choose if you had a very large *customer* relation, where only some disk blocks of the relation are accessed frequently, with other blocks rarely accessed.

12.2 Some databases use magnetic disks in a way that only sectors in outer tracks are used, while sectors in inner tracks are left unused. What might be the benefits of doing so?

12.3 Flash storage:

a. How is the flash translation table, which is used to map logical page numbers to physical page numbers, created in memory?

b. Suppose you have a 64-gigabyte flash storage system, with a 4096-byte page size. How big would the flash translation table be, assuming each page has a 32-bit address, and the table is stored as an array?

c. Suggest how to reduce the size of the translation table if very often long ranges of consecutive logical page numbers are mapped to consecutive physical page numbers.

12.4 Consider the following data and parity-block arrangement on four disks:

Telegram: @uni_k

www.konkur.in

Disk 1	Disk 2	Disk 3	Disk 4
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
⋮	⋮	⋮	⋮

The B_is represent data blocks; the P_is represent parity blocks. Parity block P_i is the parity block for data blocks B_{4i-3} to B_{4i}. What, if any, problem might this arrangement present?

12.5 A database administrator can choose how many disks are organized into a single RAID 5 array. What are the trade-offs between having fewer disks versus more disks, in terms of cost, reliability, performance during failure, and performance during rebuild?

12.6 A power failure that occurs while a disk block is being written could result in the block being only partially written. Assume that partially written blocks can be detected. An atomic block write is one where either the disk block is fully written or nothing is written (i.e., there are no partial writes). Suggest schemes for getting the effect of atomic block writes with the following RAID schemes. Your schemes should involve work on recovery from failure.

a. RAID level 1 (mirroring)

b. RAID level 5 (block interleaved, distributed parity)

12.7 Storing all blocks of a large file on consecutive disk blocks would minimize seeks during sequential file reads. Why is it impractical to do so? What do operating systems do instead, to minimize the number of seeks during sequential reads?

Exercises

12.8 List the physical storage media available on the computers you use routinely. Give the speed with which data can be accessed on each medium.

12.9 How does the remapping of bad sectors by disk controllers affect data-retrieval rates?

12.10 Operating systems try to ensure that consecutive blocks of a file are stored on consecutive disk blocks. Why is doing so very important with magnetic disks? If SSDs were used instead, is doing so still important, or is it irrelevant? Explain why.

Telegram: @uni_k

www.konkur.in

12.11 RAID systems typically allow you to replace failed disks without stopping access to the system. Thus, the data in the failed disk must be rebuilt and written to the replacement disk while the system is in operation. Which of the RAID levels yields the least amount of interference between the rebuild and ongoing disk accesses? Explain your answer.

12.12 What is scrubbing, in the context of RAID systems, and why is scrubbing important?

12.13 Suppose you have data that should not be lost on disk failure, and the application is write-intensive. How would you store the data?

Further Reading

[Hennessy et al. (2017)] is a popular textbook on computer architecture, which includes coverage of cache and memory organization.

The specifications of current-generation magnetic disk drives can be obtained from the web sites of their manufacturers, such as Hitachi, Seagate, Maxtor, and Western Digital. The specifications of current-generation SSDs can be obtained from the web sites of their manufacturers, such as Crucial, Intel, Micron, Samsung, SanDisk, Toshiba and Western Digital.

[Patterson et al. (1988)] provided early coverage of RAID levels and helped standardize the terminology. [Chen et al. (1994)] presents a survey of RAID principles and implementation.

A comprehensive coverage of RAID levels supported by most modern RAID systems, including the nested RAID levels, 10, 50 and 60, which combine RAID levels 1, 5 and 6 with striping as in RAID level 0, can be found in the "Introduction to RAID" chapter of [Cisco (2018)]. Reed-Solomon codes are covered in [Pless (1998)].

Bibliography

[Chen et al. (1994)] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, Volume 26, Number 2 (1994), pages 145–185.

[Cisco (2018)] *Cisco UCS Servers RAID Guide*. Cisco (2018).

[Hennessy et al. (2017)] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*, 6th edition, Morgan Kaufmann (2017).

[Patterson et al. (1988)] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1988), pages 109–116.

Telegram: @uni_k

www.konkur.in

[Pless (1998)] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, 3rd edition, John Wiley and Sons (1998).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Figure 12.3 is due to ©Silberschatz, Korth, and Sudarshan.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 13



Data Storage Structures

In Chapter 12 we studied the characteristics of physical storage media, focusing on magnetic disks and SSDs, and saw how to build fast and reliable storage systems using multiple disks in a RAID structure. In this chapter, we focus on the organization of data stored on the underlying storage media, and how data are accessed.

13.1 Database Storage Architecture

Persistent data are stored on non-volatile storage, which, as we saw in Chapter 12, is typically magnetic disk or SSD. Magnetic disks as well as SSDs are block structured devices, that is, data are read or written in units of a block. In contrast, databases deal with records, which are usually much smaller than a block (although in some cases records may have attributes that are very large).

Most databases use operating system files as an intermediate layer for storing records, which abstract away some details of the underlying blocks. However, to ensure efficient access, as well as to support recovery from failures (as we will see later in Chapter 19), databases must continue to be aware of blocks. Thus, in Section 13.2, we study how individual records are stored in files, taking block structure into account.

Given a set of records, the next decision lies in how to organize them in the file structure; for example, they may stored in sorted order, in the order they are created, or in an arbitrary order. Section 13.3 studies several alternative file organizations.

Section 13.4 then describes how databases organize data about the relational schemas as well as storage organization, in the data dictionary. Information in the data dictionary is crucial for many tasks, for example, to locate and retrieve records of a relation when given the name of the relation.

For a CPU to access data, it must be in main memory, whereas persistent data must be resident on non-volatile storage such as magnetic disks or SSDs. For databases that are larger than main memory, which is the usual case, data must be fetched from non-volatile storage and saved back if it is updated. Section 13.5 describes how databases use a region of memory called the database buffer to store blocks that are fetched from non-volatile storage.

Telegram: @uni_k

www.konkur.in

An approach to storing data based on storing all values of a particular column together, rather than storing all attributes of a particular row together, has been found to work very well for analytical query processing. This idea, called *column-oriented storage*, is discussed in Section 13.6.

Some applications need very fast access to data and have small enough data sizes that the entire database can fit into the main memory of a database server machine. In such cases, we can keep a copy of the entire database in memory.[1] Databases that store the entire database in memory and optimize in-memory data structures as well as query processing and other algorithms used by the database to exploit the memory residency of data are called **main-memory databases**. Storage organization in main-memory databases is discussed in Section 13.7. We note that non-volatile memory that allows direct access to individual bytes or cache lines, called *storage class memory*, is under development. Main-memory database architectures can be further optimized for such storage.

13.2 File Organization

A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks. A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.

Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer. Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created. Larger block sizes can be useful in some database applications.

A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used. We shall assume that *no record is larger than a block*. This assumption is realistic for most data-processing applications, such as our university example. There are certainly several kinds of large data items, such as images, that can be significantly larger than a block. We briefly discuss how to handle such large data items in Section 13.2.2, by storing large data items separately, and storing a pointer to the data item in the record.

In addition, we shall require that *each record is entirely contained in a single block*; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.

[1]To be safe, not only should the current database fit in memory, but there should be a reasonable certainty that the database will continue to fit in memory in the medium term future, despite

Telegram: @uni_k

www.konkur.in

In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed-length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case. Thus, we begin by considering a file of fixed-length records and consider storage of variable-length records later.

13.2.1 Fixed-Length Records

As an example, let us consider a file of *instructor* records for our university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
                      ID varchar (5);
                      name varchar(20);
                      dept_name varchar (20);
                      salary numeric (8,2);
                  end
```

Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Suppose that instead of allocating a variable amount of bytes for the attributes *ID*, *name*, and *dept_name*, we allocate the maximum number of bytes that each attribute can hold. Then, the *instructor* record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on (Figure 13.1).

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure 13.1 File containing *instructor* records.

Telegram: @uni_k

www.konkur.in

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure 13.2 File of Figure 13.1, with record 3 deleted and all records moved.

However, there are two problems with this simple approach:

1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

When a record is deleted, we could move the record that comes after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead (Figure 13.2). Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record (Figure 13.3).

It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need

Telegram: @uni_k

www.konkur.in

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Figure 13.3 File of Figure 13.1, with record 3 deleted and final record moved.

to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**. Figure 13.4 shows the file of Figure 13.1, with the free list, after records 1, 4, and 6 have been deleted.

On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.

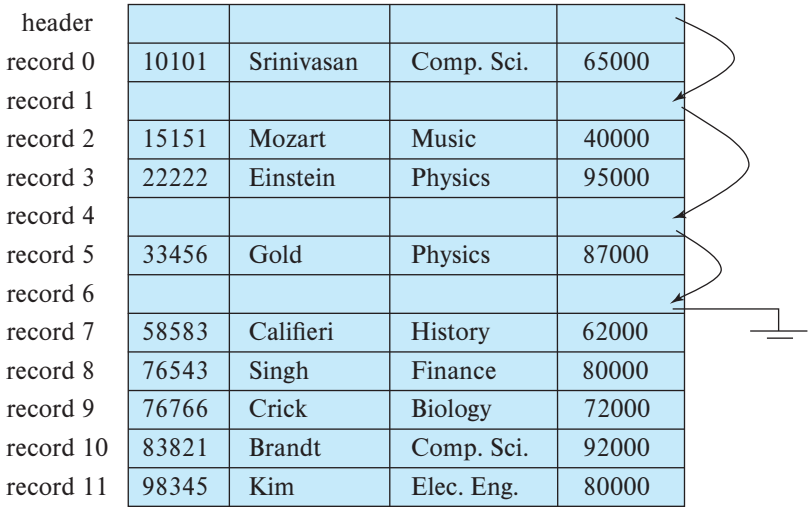


Figure 13.4 File of Figure 13.1, with free list after deletion of records 1, 4, and 6.

Telegram: @uni_k

www.konkur.in

Insertion and deletion for files of fixed-length records are simple to implement because the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable-length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

13.2.2 Variable-Length Records

Variable-length records arise in database systems due to several reasons. The most common reason is the presence of variable length fields, such as strings. Other reasons include record types that contain repeating fields such as arrays or multisets, and the presence of multiple record types within a file.

Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:

1. How to represent a single record in such a way that individual attributes can be extracted easily, even if they are of variable length
2. How to store variable-length records within a block, such that records in a block can be extracted easily

The representation of a record with variable-length attributes typically has two parts: an initial part with fixed-length information, whose structure is the same for all records of the same relation, followed by the contents of variable-length attributes. Fixed-length attributes, such as numeric values, dates, or fixed-length character strings are allocated as many bytes as required to store their value. Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (*offset*, *length*), where *offset* denotes where the data for that attribute begins within the record, and *length* is the length in bytes of the variable-sized attribute. The values for the variable-length attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length.

An example of such a record representation is shown in Figure 13.5. The figure shows an *instructor* record whose first three attributes *ID*, *name*, and *dept_name* are variable-length strings, and whose fourth attribute *salary* is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4

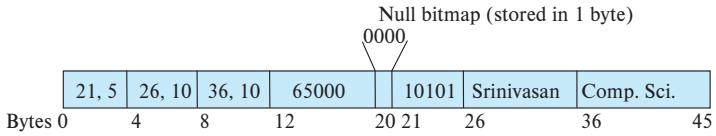


Figure 13.5 Representation of a variable-length record of the *instructor* relation.

Telegram: @uni_k

www.konkur.in

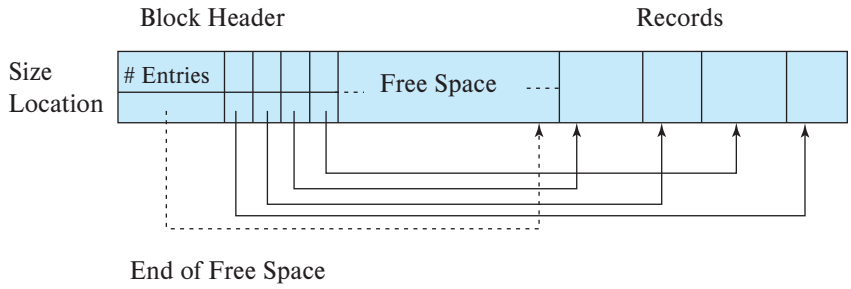


Figure 13.6 Slotted-page structure.

bytes per attribute. The *salary* attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

The figure also illustrates the use of a **null bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the *salary* value stored in bytes 12 through 19 would be ignored. Since the record has four attributes, the null bitmap for this record fits in 1 byte, although more bytes may be required with more attributes. In some representations, the null bitmap is stored at the beginning of the record, and for attributes that are null, no data (value, or offset/length) are stored at all. Such a representation would save some storage space, at the cost of extra work to extract attributes of the record. This representation is particularly useful for certain applications where records have a large number of fields, most of which are null.

We next address the problem of storing variable-length records in a block. The **slotted-page structure** is commonly used for organizing records within a block and is shown in Figure 13.6.[2] There is a header at the beginning of each block, containing the following information:

- The number of record entries in the header
- The end of free space in the block
- An array whose entries contain the location and size of each record

The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous between the final entry in the header array and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* (its size is set to −1, for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all

[2]Here, "page" is synonymous with "block."

Telegram: @uni_k

www.konkur.in

free space is again between the final entry in the header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well. Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: typical values are around 4 to 8 kilobytes.

The slotted-page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.

13.2.3 Storing Large Objects

Databases often store data that can be much larger than a disk block. For instance, an image or an audio recording may be multiple megabytes in size, while a video object may be multiple gigabytes in size. Recall that SQL supports the types **blob** and **clob**, which store binary and character large objects.

Many databases internally restrict the size of a record to be no larger than the size of a block.[3] These databases allow records to logically contain large objects, but they store the large objects separate from the other (short) attributes of records in which they occur. A (logical) pointer to the object is then stored in the record containing the large object.

Large objects may be stored either as files in a file system area managed by the database, or as file structures stored in and managed by the database. In the latter case, such in-database large objects can optionally be represented using B^+-tree file organizations, which we study in Section 14.4.1, to allow efficient access to any location within the object. B^+-tree file organizations permit us to read an entire object, or specified byte ranges in the object, as well as to insert and delete parts of the object.

However, there are some performance issues with storing very large objects in databases. The efficiency of accessing large objects via database interfaces is one concern. A second concern is the size of database backups. Many enterprises periodically create "database dumps," that is, backup copies of their databases; storing large objects in the database can result in a large increase in the size of the database dumps.

Many applications therefore choose to store very large objects, such as video data, outside of the database, in a file system. In such cases, the application may store the file name (usually a path in the file system) as an attribute of a record in the database. Storing data in files outside the database can result in file names in the database pointing to files that do not exist, perhaps because they have been deleted, which results in a form of foreign-key constraint violation. Further, database authorization controls are not applicable to data stored in the file system.

[3]This restriction helps simplify buffer management; as we see in Section 13.5, disk blocks are brought into an area of memory called the buffer before they are accessed. Records larger than a block would get split between blocks, which may be different areas of the buffer, and thus cannot be guaranteed to be in a

Telegram: @uni_k

www.konkur.in

Some databases support file system integration with the database, to ensure that constraints are satisfied (for example, deletion of files will be blocked if the database has a pointer to the file), and to ensure that access authorizations are enforced. Files can be accessed both from a file system interface and from the database SQL interface. For example, Oracle supports such integration through its SecureFiles and Database File System features.

13.3 Organization of Records in Files

So far, we have studied how records are represented in a file structure. A relation is a set of records. Given a set of records, the next question is how to organize them in a file. Several of the possible ways of organizing records in files are:

- **Heap file organization**. Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is either a single file or a set of files for each relation. Heap file organization is discussed in Section 13.3.1.
- **Sequential file organization**. Records are stored in sequential order, according to the value of a "search key" of each record. Section 13.3.2 describes this organization.
- **Multitable clustering file organization**: Generally, a separate file or set of files is used to store the records of each relation. However, in a *multitable clustering file organization*, records of several different relations are stored in the same file, and in fact in the same block within a file, to reduce the cost of certain join operations. Section 13.3.3 describes the multitable clustering file organization.
- **B+-tree file organization**. The traditional sequential file organization described in Section 13.3.2 does support ordered access even if there are insert, delete, and update operations, which may change the ordering of records. However, in the face of a large number of such operations, efficiency of ordered access suffers. We study another way of organizing records, called the *B^+-tree file organization*, in Section 14.4.1. The B^+-tree file organization is related to the B^+-tree index structure described in that chapter and can provide efficient ordered access to records even if there are a large number of insert, delete, or update operations. Further, it supports very efficient access to specific records, based on the search key.
- **Hashing file organization**. A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed. Section 14.5 describes this organization; it is closely related to the indexing structures described in that chapter.

Telegram: @uni_k

www.konkur.in

13.3.1 Heap File Organization

In a heap file organization, a record may be stored anywhere in the file corresponding to a relation. Once placed in a particular location, the record is not usually moved.[4]

When a record is inserted in a file, one option for choosing the location is to always add it at the end of the file. However, if records get deleted, it makes sense to use the space thus freed up to store new records. It is important for a database system to be able to efficiently find blocks that have free space, without having to sequentially search through all the blocks of the file.

Most databases use a space-efficient data structure called a **free-space map** to track which blocks have free space to store records. The free-space map is commonly represented by an array containing 1 entry for each block in the relation. Each entry represents a fraction f such that at least a fraction f of the space in the block is free. In PostgreSQL, for example, an entry is 1 byte, and the value stored in the entry must be divided by 256 to get the free-space fraction. The array is stored in a file, whose blocks are fetched into memory, [5] as required. Whenever a record is inserted, deleted, or changed in size, if the occupancy fraction changes enough to affect the entry value, the entry is updated in the free-space map. An example of a free-space map for a file with 16 blocks is shown below. We assume that 3 bits are used to store the occupancy fraction; the value at position i should be divided by 8 to get the free-space fraction for block i.

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6

For example, a value of 7 indicates that at least $7/8th$ of the space in the block is free.

To find a block to store a new record of a given size, the database can scan the free-space map to find a block that has enough free space to store that record. If there is no such block, a new block is allocated for the relation.

While such a scan is much faster than actually fetching blocks to find free space, it can still be very slow for large files. To further speed up the task of locating a block with sufficient free space, we can create a second-level free-space map, which has, say 1 entry for every 100 entries of the main free-space map. That 1 entry stores the maximum value amongst the 100 entries in the main free-space map that it corresponds to. The free-space map below is a second level free-space map for our earlier example, with 1 entry for every 4 entries in the main free-space map.

4	7	2	6

[4] Records may be occasionally moved, for example, if the database sorts the records of the relation; but note that even if the relation is reordered by sorting, subsequent insertions and updates may result in the records no longer being ordered.

[5] Via the database buffer, which we discuss in Section 13.5.

Telegram: @uni_k

www.konkur.in

With 1 entry for every 100 entries in the main free-space map, a scan of the second-level free space map would take only 1/100th of the time to scan the main free-space map; once a suitable entry indicating enough free space is found there, its corresponding 100 entries in the main free-space map can be examined to find a block with sufficient free space. Such a block must exist, since the second-level free-space map entry stores the maximum of the entries in the main free-space map. To deal with very large relations, we can create more levels beyond the second level, using the same idea.

Writing the free-space map to disk every time an entry in the map is updated would be very expensive. Instead, the free-space map is written periodically; as a result, the free-space map on disk may be outdated, and when a database starts up, it may get outdated data about available free space. The free-space map may, as a result, claim a block has free space when it does not; such an error will be detected when the block is fetched, and can be dealt with by a further search in the free-space map to find another block. On the other hand, the free-space map may claim that a block does not have free space when it does; generally this will not result in any problem other than unused free space. To fix any such errors, the relation is scanned periodically and the free-space map recomputed and written to disk.

13.3.2 Sequential File Organization

A **sequential file** is designed for efficient processing of records in sorted order based on some search key. A **search key** is any attribute or set of attributes; it need not be the primary key, or even a superkey. To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

Figure 13.7 shows a sequential file of *instructor* records taken from our university example. In that example, the records are stored in search-key order, using *ID* as the search key.

The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms that we shall study in Chapter 15.

It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following two rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (i.e., space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

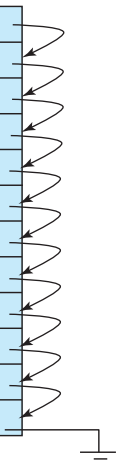


Figure 13.7 Sequential file for *instructor* records.

an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

Figure 13.8 shows the file of Figure 13.7 after the insertion of the record (32222, Verdi, Music, 48000). The structure in Figure 13.8 allows fast insertion of new records, but it forces sequential file-processing applications to process records in an order that does not match the physical order of the records.

If relatively few records need to be stored in overflow blocks, this approach works well. Eventually, however, the correspondence between search-key order and physical order may be totally lost over a period of time, in which case sequential processing will become much less efficient. At this point, the file should be **reorganized** so that it is once again physically in sequential order. Such reorganizations are costly and must be done during times when the system load is low. The frequency with which reorganizations are needed depends on the frequency of insertion of new records. In the extreme case in which insertions rarely occur, it is possible always to keep the file in physically sorted order. In such a case, the pointer field in Figure 13.7 is not needed.

The B^+-tree file organization, which we describe in Section 14.4.1, provides efficient ordered access even if there are many inserts, deletes, and updates, without requiring expensive reorganizations.

13.3.3 Multitable Clustering File Organization

Most relational database systems store each relation in a separate file, or a separate set of files. Thus, each file, and as a result, each block, stores records of only one relation, in such a design.

Telegram: @uni_k

www.konkur.in

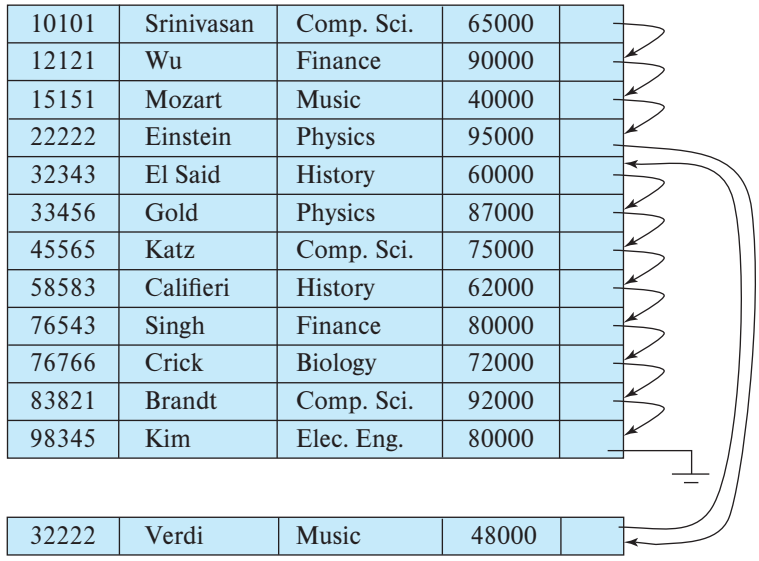


Figure 13.8 Sequential file after an insertion.

However, in some cases it can be useful to store records of more than one relation in a single block. To see the advantage of storing records of multiple relations in one block, consider the following SQL query for the university database:

select *dept_name*, *building*, *budget*, *ID*, *name*, *salary*
from *department* **natural join** *instructor*;

This query computes a join of the *department* and *instructor* relations. Thus, for each tuple of *department*, the system must locate the *instructor* tuples with the same value for *dept_name*. Ideally, these records will be located with the help of *indices*, which we shall discuss in Chapter 14. Regardless of how these records are located, however, they need to be transferred from disk into main memory. In the worst case, each record will reside on a different block, forcing us to do one block read for each record required by the query.

As a concrete example, consider the *department* and *instructor* relations of Figure 13.9 and Figure 13.10, respectively (for brevity, we include only a subset of the tuples

Comp. Sci.	Taylor	100000
Physics	Watson	70000

Figure 13.9 The *department* relation.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

Figure 13.10 The *instructor* relation.

of the relations we have used thus far). In Figure 13.11, we show a file structure designed for the efficient execution of queries involving the natural join of *department* and *instructor*. All the *instructor* tuples for a particular *dept_name* are stored near the *department* tuple for that *dept_name*. We say that the two relations are clustered on the key *dept_name*. We assume that each record contains the identifier of the relation to which it belongs, although this is not shown in Figure 13.11.

Although not depicted in the figure, it is possible to store the value of the *dept_name* attribute, which defines the clustering, only once for a group of tuples (from both relations), reducing storage overhead.

This structure allows for efficient processing of the join. When a tuple of the *department* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding *instructor* tuples are stored on the disk near the *department* tuple, the block containing the *department* tuple contains tuples of the *instructor* relation needed to process the query. If a department has so many instructors that the *instructor* records do not fit in one block, the remaining records appear on nearby blocks.

A **multitable clustering file organization** is a file organization, such as that illustrated in Figure 13.11, that stores related records of two or more relations in each block.[6] The **cluster key** is the attribute that defines which records are stored together; in our preceding example, the cluster key is *dept_name*.



Figure 13.11 Multitable clustering file structure.

[6]Note that the word *cluster* is often used to refer to a group of machines that together constitute a parallel database; that use of the word *cluster* is unrelated to the concept of multitable

Telegram: @uni_k

www.konkur.in

Although a multitable clustering file organization can speed up certain join queries, it can result in slowing processing of other types of queries. For example, in our preceding example,

```
select *
from department;
```

requires more block accesses than it did in the scheme under which we stored each relation in a separate file, since each block now contains significantly fewer *department* records. To locate efficiently all tuples of the *department* relation within a particular block, we can chain together all the records of that relation using pointers; however, the number of blocks read does not get affected by using such chains.

When multitable clustering is to be used depends on the types of queries that the database designer believes to be most frequent. Careful use of multitable clustering can produce significant performance gains in query processing.

Multitable clustering is supported by the Oracle database system. Clusters can be created by using a **create cluster** command, with a specified cluster key. An extension of the **create table** command can be used to specify that a relation is to be stored in a specific cluster, with a particular attribute used as the cluster key. Multiple relations can thus be allocated to a cluster.

13.3.4 Partitioning

Many databases allow the records in a relation to be partitioned into smaller relations that are stored separately. Such **table partitioning** is typically done on the basis of an attribute value; for example, records in a *transaction* relation in an accounting database may be partitioned by year into smaller relations corresponding to each year, such as *transaction_2018*, *transaction_2019*, and so on. Queries can be written based on the *transaction* relation but are translated into queries on the year-wise relations. Most accesses are to records of the current year and include a selection based on the year. Query optimizers can rewrite such a query to only access the smaller relation corresponding to the requested year, and they can avoid reading records corresponding to other years. For example, a query

```
select *
from transaction
where year=2019
```

would only access the relation *transaction_2019*, ignoring the other relations, while a query without the selection condition would read all the relations.

The cost of some operations, such as finding free space for a record, increase with relation size; by reducing the size of each relation, partitioning helps reduce such overheads. Partitioning can also be used to store different parts of a relation on different

Telegram: @uni_k

www.konkur.in

storage devices; for example, in the year 2019, *transaction_2018* and earlier year transactions can which are infrequently accessed could be stored on magnetic disk, while *transaction_2019* could be stored on SSD, for faster access.

13.4 Data-Dictionary Storage

So far, we have considered only the representation of the relations themselves. A relational database system needs to maintain data *about* the relations, such as the schema of the relations. In general, such "data about data" are referred to as **metadata**.

Relational schemas and other metadata about relations are stored in a structure called the **data dictionary** or **system catalog**. Among the types of information that the system must store are these:

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views defined on the database, and definitions of those views
- Integrity constraints (e.g., key constraints)

In addition, many systems keep the following data on users of the system:

- Names of users, the default schemas of the users, and passwords or other information to authenticate users
- Information about authorizations for each user

Further, the database may store statistical and descriptive data about the relations and attributes, such as the number of tuples in each relation, or the number of distinct values for each attribute.

The data dictionary may also note the storage organization (heap, sequential, hash, etc.) of relations, and the location where each relation is stored:

- If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
- If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

In Chapter 14, in which we study indices, we shall see a need to store information about each index on each of the relations:

- Name of the index

Telegram: @uni_k

www.konkur.in

- Name of the relation being indexed
- Attributes on which the index is defined
- Type of index formed

All this metadata information constitutes, in effect, a miniature database. Some database systems store such metadata by using special-purpose data structures and code. It is generally preferable to store the data about the database as relations in the database itself. By using database relations to store system metadata, we simplify the overall structure of the system and harness the full power of the database for fast access to system data.

The exact choice of how to represent system metadata by relations must be made by the system designers. We show the schema diagram of a toy data dictionary in Figure 13.12, storing part of the information mentioned above. The schema is only illustrative; real implementations store far more information than what the figure shows. Read the manuals for whichever database you use to see what system metadata it maintains.

In the metadata representation shown, the attribute *index_attributes* of the relation *Index_metadata* is assumed to contain a list of one or more attributes, which can be represented by a character string such as "*dept_name*, *building*". The *Index_metadata* relation is thus not in first normal form; it can be normalized, but the preceding representation is likely to be more efficient to access. The data dictionary is often stored in a nonnormalized form to achieve fast access.

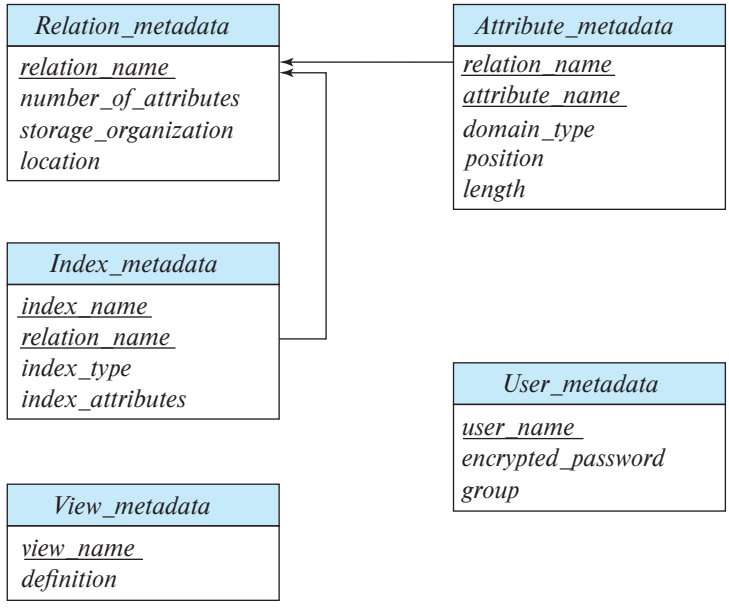


Figure 13.12 Relational schema representing part of the system metadata.

Telegram: @uni_k

www.konkur.in

Whenever the database system needs to retrieve records from a relation, it must first consult the *Relation_metadata* relation to find the location and storage organization of the relation, and then fetch records using this information.

However, the storage organization and location of the *Relation_metadata* relation itself must be recorded elsewhere (e.g., in the database code itself, or in a fixed location in the database), since we need this information to find the contents of *Relation_metadata*.

Since system metadata are frequently accessed, most databases read it from the database into in-memory data structures that can be accessed very efficiently. This is done as part of the database startup, before the database starts processing any queries.

13.5 Database Buffer

The size of main memory on servers has increased greatly over the years, and many medium-sized databases can fit in memory. However, a server has many demands on its memory, and the amount of memory it can give to a database may be much smaller than the database size even for medium-sized databases. And many large databases are much larger than the available memory on servers.

Thus, even today, database data reside primarily on disk in most databases, and they must be brought into memory to be read or updated; updated data blocks must be written back to disk subsequently.

Since data access from disk is much slower than in-memory data access, a major goal of the database system is to minimize the number of block transfers between the disk and memory. One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory. The goal is to maximize the chance that, when a block is accessed, it is already in main memory, and, thus, no disk access is required.

Since it is not possible to keep all blocks in main memory, we need to manage the allocation of the space available in main memory for the storage of blocks. The **buffer** is that part of main memory available for storage of copies of disk blocks. There is always a copy kept on disk of every block, but the copy on disk may be a version of the block older than the version in the buffer. The subsystem responsible for the allocation of buffer space is called the **buffer manager**.

13.5.1 Buffer Manager

Programs in a database system make requests (i.e., calls) on the buffer manager when they need a block from disk. If the block is already in the buffer, the buffer manager passes the address of the block in main memory to the requester. If the block is not in the buffer, the buffer manager first allocates space in the buffer for the block, throwing out some other block, if necessary, to make space for the new block. The thrown-out block is written back to disk only if it has been modified since the most recent time that it was written to the disk. Then, the buffer manager reads in the requested block from the disk to the buffer, and passes the address of the block in main memory to the

Telegram: @uni_k

www.konkur.in

requester. The internal actions of the buffer manager are transparent to the programs that issue disk-block requests.

If you are familiar with operating-system concepts, you will note that the buffer manager appears to be nothing more than a virtual-memory manager, like those found in most operating systems. One difference is that the size of the database might be larger than the hardware address space of a machine, so memory addresses are not sufficient to address all disk blocks. Further, to serve the database system well, the buffer manager must use techniques more sophisticated than typical virtual-memory management schemes:

13.5.1.1 Buffer replacement strategy

When there is no room left in the buffer, a block must be **evicted**, that is, removed, from the buffer before a new one can be read in. Most operating systems use a **least recently used** (**LRU**) scheme, in which the block that was referenced least recently is written back to disk and is removed from the buffer. This simple approach can be improved on for database applications(see Section 13.5.2).

13.5.1.2 Pinned blocks

Once a block has been brought into the buffer, a database process can read the contents of the block from the buffer memory. However, while the block is being read, if a concurrent process evicts the block and replaces it with a different block, the reader that was reading the contents of the old block will see incorrect data; if the block was being written when it was evicted, the writer would end up damaging the contents of the replacement block.

It is therefore important that before a process reads data from a buffer block, it ensures that the block will not get evicted. To do so, the process executes a **pin** operation on the block; the buffer manager never evicts a pinned block. When it has finished reading data, the process should execute an **unpin** operation, allowing the block to be evicted when required. The database code should be written carefully to avoid pinning too many blocks: if all the blocks in the buffer get pinned, no blocks can be evicted, and no other block can be brought into the buffer. If this happens, the database will be unable to carry out any further processing!

Multiple processes can read data from a block that is in the buffer. Each of them is required to execute a pin operation before accessing data, and an unpin after completing access. The block cannot be evicted until all processes that have executed a pin have then executed an unpin operation. A simple way to ensure this property is to keep a **pin count** for each buffer block. Each pin operation increments the count, and an unpin operation decrements the count. A page can be evicted only if the pin count equals 0.

13.5.1.3 Shared and Exclusive Locks on Buffer

A process that adds or deletes a tuple from a page may need to move the page contents around; during this period, no other process should read the contents of the page, since

Telegram: @uni_k

www.konkur.in

they may be inconsistent. Database buffer managers allow processes to get shared and exclusive locks on the buffer.

We will study locking in more detail in Chapter 18, but here we discuss a limited form of locking in the context of the buffer manager. The locking system provided by the buffer manager allows a database process to lock a buffer block either in shared mode or in exclusive mode before accessing the block, and to release the lock later, after the access is completed. Here are the rules for locking:

- Any number of processes may have shared locks on a block at the same time.
- Only one process is allowed to get an exclusive lock at a time, and further when a process has an exclusive lock, no other process may have a shared lock on the block. Thus, an exclusive lock can be granted only when no other process has a lock on the buffer block.
- If a process requests an exclusive lock when a block is already locked in shared or exclusive mode, the request is kept pending until all earlier locks are released.
- If a process requests a shared lock when a block is not locked, or already shared locked, the lock may be granted; however, if another process has an exclusive lock, the shared lock is granted only after the exclusive lock has been released.

Locks are acquired and released as follows:

- Before carrying out any operation on a block, a process must pin the block as we saw earlier. Locks are obtained subsequently and must be released before unpinning the block.
- Before reading data from a buffer block, a process must get a shared lock on the block. When it is done reading the data, the process must release the lock.
- Before updating the contents of a buffer block, a process must get an exclusive lock on the block; the lock must be released after the update is complete.

These rules ensure that a block cannot be updated while another process is reading it, and conversely, a block cannot be read while another process is updating it. These rules are required for safety of buffer access; however, to protect a database system from problems due to concurrent access, these steps are not sufficient: further steps need to be taken. These are discussed further in Chapter 17 and Chapter 18.

13.5.1.4 Output of blocks

It is possible to output a block only when the buffer space is needed for another block. However, it makes sense to not wait until the buffer space is needed, but to rather write out updated blocks ahead of such a need. Then, when space is required in the buffer,

Telegram: @uni_k

www.konkur.in

a block that has already been written out can be evicted, provided it is not currently pinned.

However, for the database system to be able to recover from crashes (Chapter 19), it is necessary to restrict those times when a block may be written back to disk. For instance, most recovery systems require that a block should not be written to disk while an update on the block is in progress. To enforce this requirement, a process that wishes to write the block to disk must obtain a shared lock on the block.

Most databases have a process that continually detects updated blocks and writes them back to disk.

13.5.1.5 Forced output of blocks

There are situations in which it is necessary to write a block to disk, to ensure that certain data on disk are in a consistent state. Such a write is called a **forced output** of a block. We shall see the reason for forced output in Chapter 19.

Memory contents and thus buffer contents are lost in a crash, whereas data on disk (usually) survive a crash. Forced output is used in conjunction with a logging mechanism to ensure that when a transaction that has performed updates commits, enough data has been written to disk to ensure the updates of the transaction are not lost. How exactly this is done is covered in detail in Chapter 19.

13.5.2 Buffer-Replacement Strategies

The goal of a replacement strategy for blocks in the buffer is to minimize accesses to the disk. For general-purpose programs, it is not possible to predict accurately which blocks will be referenced. Therefore, operating systems use the past pattern of block references as a predictor of future references. The assumption generally made is that blocks that have been referenced recently are likely to be referenced again. Therefore, if a block must be replaced, the least recently referenced block is replaced. This approach is called the **least recently used** (**LRU**) block-replacement scheme.

LRU is an acceptable replacement scheme in operating systems. However, a database system is able to predict the pattern of future references more accurately than an operating system. A user request to the database system involves several steps. The database system is often able to determine in advance which blocks will be needed by looking at each of the steps required to perform the user-requested operation. Thus, unlike operating systems, which must rely on the past to predict the future, database systems may have information regarding at least the short-term future.

To illustrate how information about future block access allows us to improve the LRU strategy, consider the processing of the SQL query:

```
select *
from instructor natural join department;
```

Telegram: @uni_k

www.konkur.in

Assume that the strategy chosen to process this request is given by the pseudocode program shown in Figure 13.13. (We shall study other, more efficient, strategies in Chapter 15.)

Assume that the two relations of this example are stored in separate files. In this example, we can see that, once a tuple of *instructor* has been processed, that tuple is not needed again. Therefore, once processing of an entire block of *instructor* tuples is completed, that block is no longer needed in main memory, even though it has been used recently. The buffer manager should be instructed to free the space occupied by an *instructor* block as soon as the final tuple has been processed. This buffer-management strategy is called the **toss-immediate** strategy.

Now consider blocks containing *department* tuples. We need to examine every block of *department* tuples once for each tuple of the *instructor* relation. When processing of a *department* block is completed, we know that that block will not be accessed again until all other *department* blocks have been processed. Thus, the most recently used *department* block will be the final block to be re-referenced, and the least recently used *department* block is the block that will be referenced next. This assumption set is the exact opposite of the one that forms the basis for the LRU strategy. Indeed, the optimal strategy for block replacement for the above procedure is the **most recently used** (**MRU**) strategy. If a *department* block must be removed from the buffer, the MRU strategy chooses the most recently used block (blocks are not eligible for replacement while they are being used).

```
for each tuple i of instructor do
    for each tuple d of department do
        if i[dept_name] = d[dept_name]
        then begin
                let x be a tuple defined as follows:
                x[ID] := i[ID]
                x[dept_name] := i[dept_name]
                x[name] := i[name]
                x[salary] := i[salary]
                x[building] := d[building]
                x[budget] := d[budget]
                include tuple x as part of result of instructor ⋈ department
             end
    end
end
```

Figure 13.13 Procedure for computing join.

Telegram: @uni_k

www.konkur.in

For the MRU strategy to work correctly for our example, the system must pin the *department* block currently being processed. After the final *department* tuple has been processed, the block is unpinned, and it becomes the most recently used block.

In addition to using knowledge that the system may have about the request being processed, the buffer manager can use statistical information about the probability that a request will reference a particular relation. For example, the data dictionary, which we saw in Section 13.4, is one of the most frequently accessed parts of the database, since the processing of every query needs to access the data dictionary. Thus, the buffer manager should try not to remove data-dictionary blocks from main memory, unless other factors dictate that it do so. In Chapter 14, we discuss indices for files. Since an index for a file may be accessed more frequently than the file itself, the buffer manager should, in general, not remove index blocks from main memory if alternatives are available.

The ideal database block-replacement strategy needs knowledge of the database operations—both those being performed and those that will be performed in the future. No single strategy is known that handles all the possible scenarios well. Indeed, a surprisingly large number of database systems use LRU, despite that strategy's faults. The practice questions and exercises explore alternative strategies.

The strategy that the buffer manager uses for block replacement is influenced by factors other than the time at which the block will be referenced again. If the system is processing requests by several users concurrently, the concurrency-control subsystem (Chapter 18) may need to delay certain requests, to ensure preservation of database consistency. If the buffer manager is given information from the concurrency-control subsystem indicating which requests are being delayed, it can use this information to alter its block-replacement strategy. Specifically, blocks needed by active (nondelayed) requests can be retained in the buffer at the expense of blocks needed by the delayed requests.

The crash-recovery subsystem (Chapter 19) imposes stringent constraints on block replacement. If a block has been modified, the buffer manager is not allowed to write back the new version of the block in the buffer to disk, since that would destroy the old version. Instead, the block manager must seek permission from the crash-recovery subsystem before writing out a block. The crash-recovery subsystem may demand that certain other blocks be force-output before it grants permission to the buffer manager to output the block requested. In Chapter 19, we define precisely the interaction between the buffer manager and the crash-recovery subsystem.

13.5.3 Reordering of Writes and Recovery

Database buffers allow writes to be performed in-memory and output to disk at a later time, possibly in an order different from the order in which the writes were performed. File systems, too, routinely reorder write operations. However, such reordering can lead to inconsistent data on disk in the event of a system crash.

Telegram: @uni_k

www.konkur.in

To understand the problem in the context of a file system, suppose that a file system uses a linked list to track which blocks are part of a file. Suppose also that it inserts a new node at the end of the list by first writing the data for the new node, then updating the pointer from the previous node. Suppose further that the writes were reordered, so the pointer was updated first, and the system crashes before the new node is written. The contents of the node would then be whatever happened to be on that disk earlier, resulting in a corrupted data structure.

To deal with the possibility of such data structure corruption, earlier-generation file systems had to perform a *file system consistency check* on system restart, to ensure that the data structures were consistent. And if they were not, extra steps had to be taken to restore them to consistency. These checks resulted in long delays in system restart after a crash, and the delays became worse as disk systems grew to higher capacities.

The file system can avoid inconsistencies in many cases if it writes updates to metadata in a carefully chosen order. But doing so would mean that optimizations such as disk arm scheduling cannot be done, affecting the efficiency of the update. If a non-volatile write buffer were available, it could be used to perform the writes in order to non-volatile RAM and later reorder the writes when writing them to disk.

However, most disks do not come with a non-volatile write buffer; instead, modern file systems assign a disk for storing a log of the writes in the order that they are performed. Such a disk is called a **log disk**. For each write, the log contains the block number to be written to, and the data to be written, in the order in which the writes were performed. All access to the log disk is sequential, essentially eliminating seek time, and several consecutive blocks can be written at once, making writes to the log disk several times faster than random writes. As before, the data have to be written to their actual location on disk as well, but the write to the actual location can be done later; the writes can be reordered to minimize disk-arm movement.

If the system crashes before some writes to the actual disk location have been completed, when the system comes back up it reads the log disk to find those writes that had not been completed and carries them out then. After the writes have been performed, the records are deleted from the log disk.

File systems that support log disks as above are called **journaling file systems**. Journaling file systems can be implemented even without a separate log disk, keeping data and the log on the same disk. Doing so reduces the monetary cost at the expense of lower performance.

Most modern file systems implement journaling and use the log disk when writing file system metadata such as file allocation information. Journaling file systems allow quick restart without the need for such file system consistency checks.

However, writes performed by applications are usually not written to the log disk. Database systems instead implement their own forms of logging, which we study in Chapter 19, to ensure that the contents of a database can be safely recovered in the event of a failure, even if writes were reordered.

Telegram: @uni_k

www.konkur.in

13.6 Column-Oriented Storage

Databases traditionally store all attributes of a tuple together in a record, and tuples are stored in a file as we have just seen. Such a storage layout is referred to as a *row-oriented storage*.

In contrast, in **column-oriented storage**, also called a **columnar storage**, each attribute of a relation is stored separately, with values of the attribute from successive tuples stored at successive positions in the file. Figure 13.14 shows how the *instructor* relation would be stored in column-oriented storage, with each attribute stored separately.

In the simplest form of column-oriented storage, each attribute is stored in a separate file. Further, each file is *compressed*, to reduce its size. (We discuss more complex schemes that store columns consecutively in a single file later in this section.)

If a query needs to access the entire contents of the ith row of a table, the values at the ith position in each of the columns are retrieved and used to reconstruct the row. Column-oriented storage thus has the drawback that fetching multiple attributes of a single tuple requires multiple I/O operations. Thus, it is not suitable for queries that fetch multiple attributes from a few rows of a relation.

However, column-oriented storage is well suited for data analysis queries, which process many rows of a relation, but often only access some of the attributes. The reasons are as follows:

- **Reduced I/O**. When a query needs to access only a few attributes of a relation with a large number of attributes, the remaining attributes need not be fetched from disk into memory. In contrast, in row-oriented storage, irrelevant attributes are fetched into memory from disk. The reduction in I/O can lead to significant reduction in query execution cost.



Figure 13.14 Columnar representation of the *instructor* relation.

Telegram: @uni_k

www.konkur.in

- **Improved CPU cache performance**. When the query processor fetches the contents of a particular attribute, with modern CPU architectures multiple consecutive bytes, called a cache line, are fetched from memory to CPU cache. If these bytes are accessed later, access is much faster if they are in cache than if they have to be fetched from main memory. However, if these adjacent bytes contain values for attributes that are not be needed by the query, fetching them into cache wastes memory bandwidth and uses up cache space that could have been used for other data. Column-oriented storage does not suffer from this problem, since adjacent bytes are from the same column, and data analysis queries usually access all these values consecutively.
- **Improved compression**. Storing values of the same type together significantly increases the effectiveness of compression, when compared to compressing data stored in row format; in the latter case, adjacent attributes are of different types, reducing the efficiency of compression. Compression significantly reduces the time taken to retrieve data from disk, which is often the highest-cost component for many queries. If the compressed files are stored in memory, the in-memory storage space is also reduced correspondingly, which is particularly important since main memory is significantly more expensive than disk storage.
- **Vector processing**. Many modern CPU architectures support **vector processing**, which allows a CPU operation to be applied in parallel on a number of elements of an array. Storing data columnwise allows vector processing of operations such as comparing an attribute with a constant, which is important for applying selection conditions on a relation. Vector processing can also be used to compute an aggregate of multiple values in parallel, instead of aggregating the values one at a time.

As a result of these benefits, column-oriented storage is increasingly used in data-warehousing applications, where queries are primarily data analysis queries. It should be noted that indexing and query processing techniques need to be carefully designed to get the performance benefits of column-oriented storage. We outline indexing and query processing techniques based on bitmap representations, which are well suited to column-oriented storage, in Section 14.9; further details are provided in Section 24.3.

Databases that use column-oriented storage are referred to as **column stores**, while databases that use row-oriented storage are referred to as **row stores**.

It should be noted that column-oriented storage does have several drawbacks, which make them unsuitable for transaction processing.

- **Cost of tuple reconstruction**. As we saw earlier, reconstructing a tuple from the individual columns can be expensive, negating the benefits of columnar representation if many columns need to be reconstructed. While tuple reconstruction is common in transaction-processing applications, data analysis applications usually

Telegram: @uni_k

www.konkur.in

output only a few columns out of many that are stored in "fact tables" in data warehouses.

- **Cost of tuple deletion and update**. Deleting or updating a single tuple in a compressed representation would require rewriting the entire sequence of tuples that are compressed as one unit. Since updates and deletes are common in transaction-processing applications, column-oriented storage would result in a high cost for these operations if a large number of tuples were compressed as one unit.

  In contrast, data-warehousing systems typically do not support updates to tuples, and instead support only insert of new tuples and bulk deletes of a large number of old tuples at a time. Inserts are done at the end of the relation representation, that is, new tuples are appended to the relation. Since small deletes and updates do not occur in a data warehouse, large sequences of attribute values can be stored and compressed together as one unit, allowing for better compression than with small sequences.

- **Cost of decompression**. Fetching data from a compressed representation requires *decompression*, which in the simplest compressed representations requires reading all the data from the beginning of a file. Transaction processing queries usually only need to fetch a few records; sequential access is expensive in such a scenario, since many irrelevant records may have to be decompressed to access a few relevant records.

Since data analysis queries tend to access many consecutive records, the time spent on decompression is typically not wasted. However, even data analysis queries do not need to access records that fail selection conditions, and attributes of such records should be skipped to reduce disk I/O.

To allow skipping of attribute values from such records, compressed representations for column stores allow decompression to start at any of a number of points in the file, skipping earlier parts of the file. This could be done by starting compression afresh after every 10,000 values (for example). By keeping track of where in the file the data start for each group of 10,000 values, it is possible to access the ith value by going to the start of the group $\lfloor i/10000 \rfloor$ and starting decompression from there.

ORC and Parquet are columnar file representations used in many big-data processing applications. In ORC, a row-oriented representation is converted to column-oriented representation as follows: A sequence of tuples occupying several hundred megabytes is broken up into a columnar representation called a **stripe**. An ORC file contains several such stripes, with each stripe occupying around 250 megabytes.

Figure 13.15 illustrates some details of the ORC file format. Each stripe has index data followed by row data. The row data area stores a compressed representation of the sequence of value for the first column, followed by the compressed representation of the second column, and so on. The index data region of a stripe stores for each attribute the starting point within the stripe for each group of (say) 10,000 values of

Telegram: @uni_k

www.konkur.in

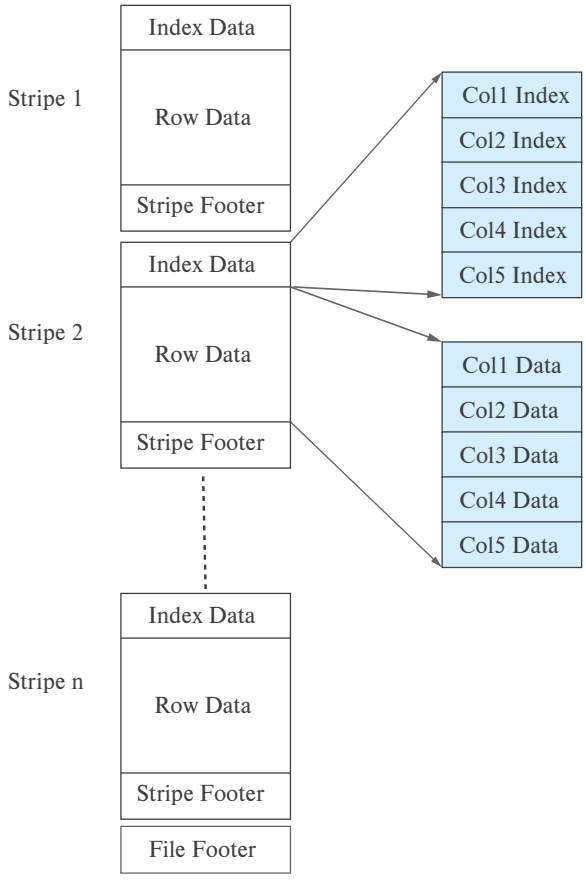


Figure 13.15 Columnar data representation in the ORC file format.

that attribute.[7] The index is useful for quick access to a desired tuple or sequence of tuples; the index also allows queries containing selections to skip groups of tuples if the query determines that no tuple in those groups satisfies the selections. ORC files store several other pieces of information in the stripe footer and file footer, which we skip here.

Some column-store systems allow groups of columns that are often accessed together to be stored together, instead of breaking up each column into a different file. Such systems thus allow a spectrum of choices that range from pure column-oriented storage, where every column is stored separately, to pure row-oriented storage, where all columns are stored together. The choice of which attributes to store together depends on the query workload.

[7] ORC files have some other information that we ignore here.

Telegram: @uni_k

www.konkur.in

Some of the benefits of column-oriented storage can be obtained even in a row-oriented storage system by logically decomposing a relation into multiple relations. For example, the *instructor* relation could be decomposed into three relations, containing (*ID*, *name*), (*ID*, *dept_name*) and (*ID*, *salary*), respectively. Then, queries that access only the name do not have to fetch the *dept_name* and *salary* attributes. However, in this case the same *ID* attribute occurs in three tuples, resulting in wasted space.

Some database systems use a column-oriented representation for data within a disk block, without using compression.[8] Thus, a block contains data for a set of tuples, and all attributes for that set of tuples are stored in the same block. Such a scheme is useful in transaction-processing systems, since retrieving all attribute values does not require multiple disk accesses. At the same time, using column-oriented storage within the block provides the benefits of more efficient memory access and cache usage, as well as the potential for using vector processing on the data. However, this scheme does not allow irrelevant disk blocks to be skipped when only a few attributes are retrieved, nor does it give the benefits of compression. Thus, it represents a point in the space between pure row-oriented storage and pure column-oriented storage.

Some databases, such as SAP HANA support two underlying storage systems, one a row-oriented one designed for transaction processing, and the second a column-oriented one, designed for data analysis. Tuples are normally created in the row-oriented store but are later migrated to the column-oriented store when they are no longer likely to be accessed in a row-oriented manner. Such systems are called **hybrid row/column stores**.

In other cases, applications store transactional data in a row-oriented store, but copy data periodically (e.g., once a day or a few times a day) to a data warehouse, which may use a column-oriented storage system.

Sybase IQ was one of the early products to use column-oriented storage, but there are now several research projects and companies that have developed database systems based on column stores, including C-Store, Vertica, MonetDB, Vectorwise, among others. See Further Reading at the end of the chapter for more details.

13.7 Storage Organization in Main-Memory Databases

Today, main-memory sizes are large enough, and main memory is cheap enough, that many organizational databases fit entirely in memory. Such large main memories can be used by allocating a large amount of memory to the database buffer, which will allow the entire database to be loaded into buffer, avoiding disk I/O operations for reading data; updated blocks still have to be written back to disk for persistence. Thus, such a setup would provide much better performance than one where only part of the database can fit in the buffer.

[8] Compression can be applied to data in a disk block, but accessing them requires decompression, and the decompressed data may no longer fit in a block. Significant changes need to be made to the database code, including buffer management, to handle such issues.

Telegram: @uni_k

www.konkur.in

However, if the entire database fits in memory, performance can be improved significantly by tailoring the storage organization and database data structures to exploit the fact that data are fully in memory. A **main-memory database** is a database where all data reside in memory; main-memory database systems are typically designed to optimize performance by making use of this fact. In particular, they do away entirely with the buffer manager.

As an example of optimizations that can be done with memory-resident data, consider the cost of accessing a record, given a record pointer. With disk-based databases, records are stored in blocks, and pointers to records consist of a block identifier and an offset or slot number within the block. Following such a record pointer requires checking if the block is in the buffer (usually done by using an in-memory hash index), and if it is, finding where in the buffer it is located. If it is not in buffer, it has to be fetched. All these actions take a significant number of CPU cycles.

In contrast, in a main-memory database, it is possible to keep direct pointers to records in memory, and accessing a record is just an in-memory pointer traversal, which is a very efficient operation. This is possible as long as records are not moved around. Indeed, one reason for such movement, namely loading into buffer and eviction from buffer, is no longer an issue.

If records are stored in a slotted-page structure within a block, records may move within a block as other records are deleted or resized. Direct pointers to records are not possible in that case, although records can be accessed with one level of indirection through the entries in the slotted page header. Locking of the block may be required to ensure that a record does not get moved while another process is reading its data. To avoid these overheads, many main-memory databases do not use a slotted-page structure for allocating records. Instead they directly allocate records in main memory, and ensure that records never get moved due to updates to other records. However, a problem with direct allocation of records is that memory may get fragmented if variable sized records are repeatedly inserted and deleted. The database must ensure that main memory does not get fragmented over time, either by using suitably designed memory management schemes or by periodically performing compaction of memory; the latter scheme will result in record movement, but it can be done without acquiring locks on blocks.

If a column-oriented storage scheme is used in main memory, all the values of a column can be stored in consecutive memory locations. However, if there are appends to the relation, ensuring contiguous allocation would require existing data be reallocated. To avoid this overhead, the logical array for a column may be divided into multiple physical arrays. An indirection table stores pointers to all the physical arrays. This scheme is depicted in Figure 13.16. To find the ith element of a logical array, the indirection table is used to locate the physical array containing the ith element, and then an appropriate offset is computed and looked up within that physical array.

There are other ways in which processing can be optimized with main-memory databases, as we shall see in later chapters.

Telegram: @uni_k

www.konkur.in

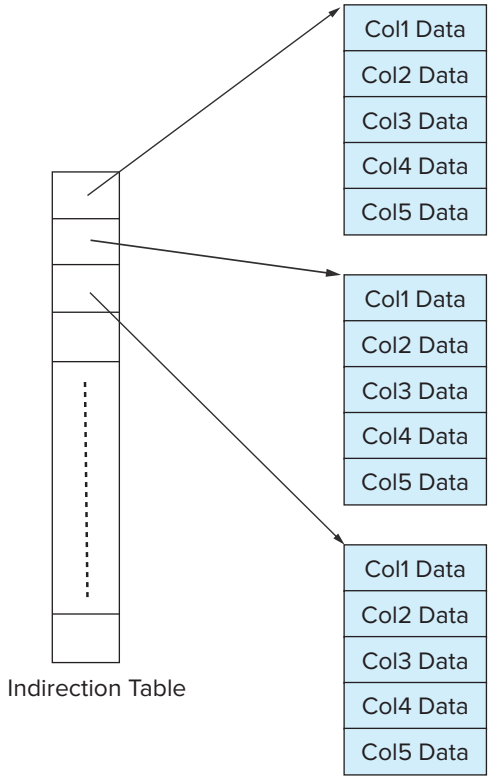


Figure 13.16 In-memory columnar data representation.

13.8 Summary

- We can organize a file logically as a sequence of records mapped onto disk blocks. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure files so that they can accommodate multiple lengths for records. The slotted-page method is widely used to handle varying-length records within a disk block.
- Since data are transferred between disk storage and main memory in units of a block, it is worthwhile to assign file records to blocks in such a way that a single block contains related records. If we can access several of the records we want with only one block access, we save disk accesses. Since disk accesses are usually the bottleneck in the performance of a database system, careful assignment of records to blocks can pay significant performance dividends.
- The data dictionary, also referred to as the system catalog, keeps track of metadata, that is, data about data, such as relation names, attribute names and types, storage information, integrity constraints, and user information.

Telegram: @uni_k

www.konkur.in

- One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory. Since it is not possible to keep all blocks in main memory, we need to manage the allocation of the space available in main memory for the storage of blocks. The *buffer* is that part of main memory available for storage of copies of disk blocks. The subsystem responsible for the allocation of buffer space is called the *buffer manager*.
- Column-oriented storage systems provide good performance for many data warehousing applications.

Review Terms

- File Organization
  - File
  - Blocks
- Fixed-length records
- File header
- Free list
- Variable-length records
- Null bitmap
- Slotted-page structure
- Large objects
- Organization of records
  - Heap file organization
  - Sequential file organization
  - Multitable clustering file organization
  - B^+-tree file organizations
  - Hashing file organization
- Free-space map
- Sequential file
- Search key
- Cluster key
- Table partitioning
- Data-dictionary storage
  - Metadata
  - Data dictionary
  - System catalog
- Database buffer
  - Buffer manager
  - Pinned blocks
  - Evicted blocks
  - Forced output of blocks
  - Shared and exclusive locks
- Buffer-replacement strategies
  - Least recently used (LRU)
  - Toss-immediate
  - Most recently used (MRU)
- Output of blocks
- Forced output of blocks
- Log disk
- Journaling file systems
- Column-oriented storage
  - Columnar storage
  - Vector processing
  - Column stores
  - Row stores
  - Stripe
  - Hybrid row/column storage
- Main-memory database

Telegram: @uni_k

www.konkur.in

Practice Exercises

13.1 Consider the deletion of record 5 from the file of Figure 13.3. Compare the relative merits of the following techniques for implementing the deletion:

a. Move record 6 to the space occupied by record 5, and move record 7 to the space occupied by record 6.

b. Move record 7 to the space occupied by record 5.

c. Mark record 5 as deleted, and move no records.

13.2 Show the structure of the file of Figure 13.4 after each of the following steps:

a. Insert (24556, Turnamian, Finance, 98000).

b. Delete record 2.

c. Insert (34556, Thompson, Music, 67000).

13.3 Consider the relations *section* and *takes*. Give an example instance of these two relations, with three sections, each of which has five students. Give a file structure of these relations that uses multitable clustering.

13.4 Consider the bitmap representation of the free-space map, where for each block in the file, two bits are maintained in the bitmap. If the block is between 0 and 30 percent full the bits are 00, between 30 and 60 percent the bits are 01, between 60 and 90 percent the bits are 10, and above 90 percent the bits are 11. Such bitmaps can be kept in memory even for quite large files.

a. Outline two benefits and one drawback to using two bits for a block, instead of one byte as described earlier in this chapter.

b. Describe how to keep the bitmap up to date on record insertions and deletions.

c. Outline the benefit of the bitmap technique over free lists in searching for free space and in updating free space information.

13.5 It is important to be able to quickly find out if a block is present in the buffer, and if so where in the buffer it resides. Given that database buffer sizes are very large, what (in-memory) data structure would you use for this task?

13.6 Suppose your university has a very large number of *takes* records, accumulated over many years. Explain how table partitioning can be done on the *takes* relation, and what benefits it could offer. Explain also one potential drawback of the technique.

Telegram: @uni_k

www.konkur.in

13.7 Give an example of a relational-algebra expression and a query-processing strategy in each of the following situations:

a. MRU is preferable to LRU.

b. LRU is preferable to MRU.

13.8 PostgreSQL normally uses a small buffer, leaving it to the operating system buffer manager to manage the rest of main memory available for file system buffering. Explain (a) what is the benefit of this approach, and (b) one key limitation of this approach.

Exercises

13.9 In the variable-length record representation, a null bitmap is used to indicate if an attribute has the null value.

a. For variable-length fields, if the value is null, what would be stored in the offset and length fields?

b. In some applications, tuples have a very large number of attributes, most of which are null. Can you modify the record representation such that the only overhead for a null attribute is the single bit in the null bitmap?

13.10 Explain why the allocation of records to blocks affects database-system performance significantly.

13.11 List two advantages and two disadvantages of each of the following strategies for storing a relational database:

a. Store each relation in one file.

b. Store multiple relations (perhaps even the entire database) in one file.

13.12 In the sequential file organization, why is an overflow *block* used even if there is, at the moment, only one overflow record?

13.13 Give a normalized version of the *Index_metadata* relation, and explain why using the normalized version would result in worse performance.

13.14 Standard buffer managers assume each block is of the same size and costs the same to read. Consider a buffer manager that, instead of LRU, uses the rate of reference to objects, that is, how often an object has been accessed in the last n seconds. Suppose we want to store in the buffer objects of varying sizes, and varying read costs (such as web pages, whose read cost depends on the site from which they are fetched). Suggest how a buffer manager may choose which block to evict from the buffer.

Telegram: @uni_k

www.konkur.in

Further Reading

[Hennessy et al. (2017)] is a popular textbook on computer architecture, which includes coverage of hardware aspects of translation look-aside buffers, caches, and memory-management units.

The storage structure of specific database systems, such as IBM DB2, Oracle, Microsoft SQL Server, and PostgreSQL are documented in their respective system manuals, which are available online.

Algorithms for buffer management in database systems, along with a performance evaluation, were presented by [Chou and Dewitt (1985)]. Buffer management in operating systems is discussed in most operating-system texts, including in [Silberschatz et al. (2018)].

[Abadi et al. (2008)] presents a comparison of column-oriented and row-oriented storage, including issues related to query processing and optimization.

Sybase IQ, developed in the mid 1990s, was the first commercially successful column-oriented database, designed for analytics. MonetDB and C-Store were column-oriented databases developed as academic research projects. The Vertica column-oriented database is a commercial database that grew out of C-Store, while VectorWise is a commercial database that grew out of MonetDB. As its name suggests, VectorWise supports vector processing of data, and as a result supports very high processing rates for many analytical queries. [Stonebraker et al. (2005)] describe C-Store, while [Idreos et al. (2012)] give an overview of the MonetDB project and [Zukowski et al. (2012)] describes Vectorwise.

The ORC and Parquet columnar file formats were developed to support compressed storage of data for big-data applications that run on the Apache Hadoop platform.

Bibliography

[Abadi et al. (2008)] D. J. Abadi, S. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How Different Are They Really?", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2008), pages 967–980.

[Chou and Dewitt (1985)] H. T. Chou and D. J. Dewitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", In *Proc. of the International Conf. on Very Large Databases* (1985), pages 127–141.

[Hennessy et al. (2017)] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*, 6th edition, Morgan Kaufmann (2017).

[Idreos et al. (2012)] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures", *IEEE Data Engineering Bulletin*, Volume 35, Number 1 (2012), pages 40–45.

[Silberschatz et al. (2018)] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th edition, John Wiley and Sons

Telegram: @uni_k

www.konkur.in

[Stonebraker et al. (2005)] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-Store: A Column-oriented DBMS", In *Proc. of the International Conf. on Very Large Databases* (2005), pages 553–564.

[Zukowski et al. (2012)] M. Zukowski, M. van de Wiel, and P. A. Boncz, "Vectorwise: A Vectorized Analytical DBMS", In *Proc. of the International Conf. on Data Engineering* (2012), pages 1349–1350.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 14



Indexing

Many queries reference only a small proportion of the records in a file. For example, a query like "Find all instructors in the Physics department" or "Find the total number of credits earned by the student with *ID* 22201" references only a fraction of the *instructor* or *student* records. It is inefficient for the system to read every tuple in the *instructor* relation to check if the *dept_name* value is "Physics". Likewise, it is inefficient to read the entire *student* relation just to find the one tuple for the *ID* "22201". Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

14.1 Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.

Database-system indices play the same role as book indices in libraries. For example, to retrieve a *student* record given an *ID*, the database system would look up an index to find on which disk block[1] the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record.

Indices are critical for efficient processing of queries in databases. Without indices, every query would end up reading the entire contents of every relation that it uses; doing so would be unreasonably expensive for queries that only fetch a few records, for example, a single *student* record, or the records in the *takes* relation corresponding to a single student.

[1] As in earlier chapters, we use the term *disk* to refer to persistent storage devices, such as magnetic disks and solid-state drives.

Telegram: @uni_k

www.konkur.in

Implementing an index on the *student* relation by keeping a sorted list of students' *ID* would not work well on very large databases, since (i) the index would itself be very big, (ii) even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming, and (iii) updating a sorted list as students are added or removed from the database can be very expensive. Instead, more sophisticated indexing techniques are used in database systems. We shall discuss several of these techniques in this chapter.

There are two basic kinds of indices:

- **Ordered indices**. Based on a sorted ordering of the values.
- **Hash indices**. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

We shall consider several techniques for ordered indexing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

- **Access types**: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead**: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

We often want to have more than one index for a file. For example, we may wish to search for a book by author, by subject, or by title.

An attribute or set of attributes used to look up records in a file is called a **search key**. Note that this definition of *key* differs from that used in *primary key*, *candidate key*, and *superkey*. This duplicate meaning for *key* is (unfortunately) well established in practice. Using our notion of a search key, we see that if there are several indices on a file, there are several search keys.

Telegram: @uni_k

www.konkur.in

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

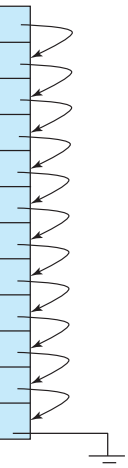


Figure 14.1 Sequential file for *instructor* records.

14.2 Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an **ordered index** stores the values of the search keys in sorted order and associates with each search key the records that contain it.

The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as the Dewey decimal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file. Clustering indices are also called **primary indices**; the term *primary index* may appear to denote an index on a primary key, but such indices can in fact be built on any search key. The search key of a clustering index is often the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary indices**. The terms "clustered" and "nonclustered" are often used in place of "clustering" and "nonclustering."

In Section 14.2.1 through Section 14.2.3, we assume that all files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called **index-sequential files**. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records. In Section 14.2.4 we cover secondary indices.

Figure 14.1 shows a sequential file of *instructor* records taken from our university example. In the example of Figure 14.1, the records are stored in sorted order of instructor *ID*, which is used as the search key.

Telegram: @uni_k

www.konkur.in

14.2.1 Dense and Sparse Indices

An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

There are two types of ordered indices that we can use:

- **Dense index**: In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key.

  In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

- **Sparse index**: In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key; that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry and follow the pointers in the file until we find the desired record.

Figure 14.2 and Figure 14.3 show dense and sparse indices, respectively, for the *instructor* file. Suppose that we are looking up the record of instructor with *ID* "22222". Using the dense index of Figure 14.2, we follow the pointer directly to the desired record. Since *ID* is a primary key, there exists only one such record and the search is complete. If we are using the sparse index (Figure 14.3), we do not find an index entry for "22222". Since the last entry (in numerical order) before "22222" is "10101", we follow that pointer. We then read the *instructor* file in sequential order until we find the desired record.

Consider a (printed) dictionary. The header of each page lists the first word alphabetically on that page. The words at the top of each page of the book index together form a sparse index on the contents of the dictionary pages.

As another example, suppose that the search-key value is not a primary key. Figure 14.4 shows a dense clustering index for the *instructor* file with the search key being *dept_name*. Observe that in this case the *instructor* file is sorted on the search key *dept_name*, instead of *ID*, otherwise the index on *dept_name* would be a nonclustering index. Suppose that we are looking up records for the History department. Using the dense index of Figure 14.4, we follow the pointer directly to the first History record. We process this record and follow the pointer in that record to locate the next record in

Telegram: @uni_k

www.konkur.in

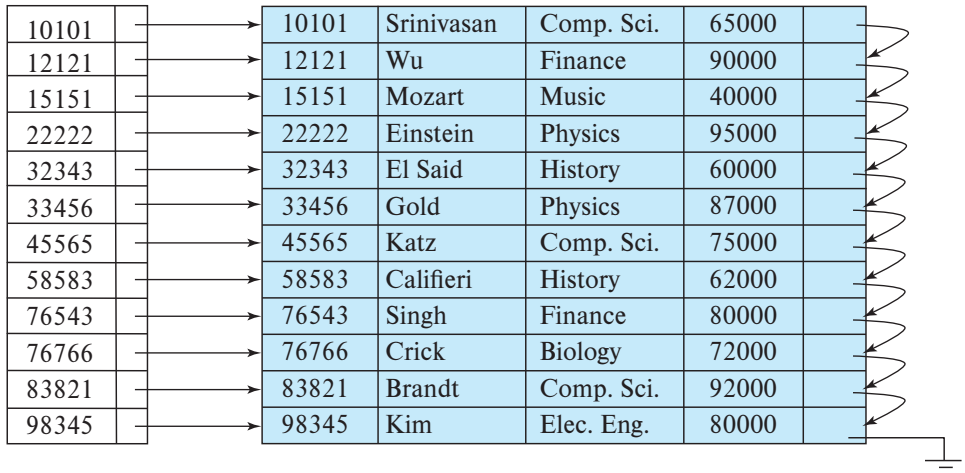


Figure 14.2 Dense index.

search-key (*dept_name*) order. We continue processing records until we encounter a record for a department other than History.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per

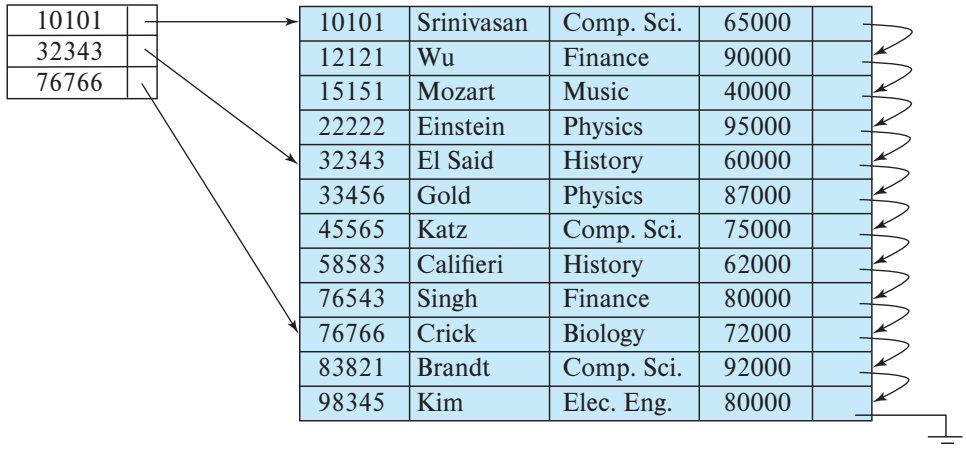


Figure 14.3 Sparse index.

Telegram: @uni_k

www.konkur.in

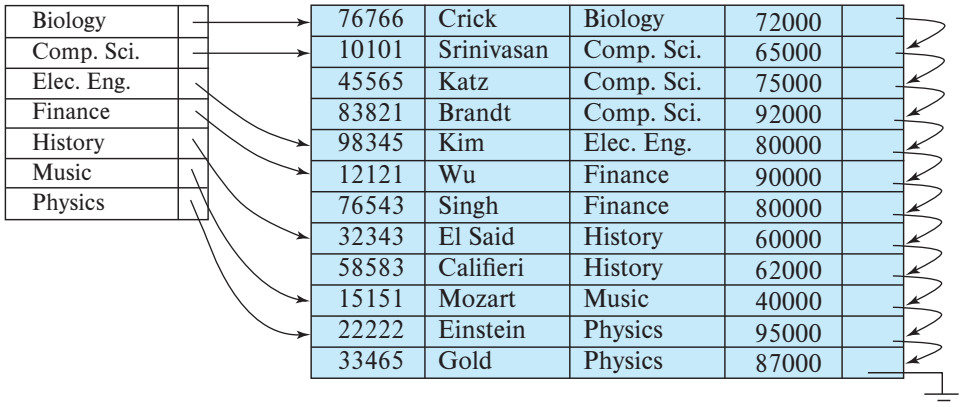


Figure 14.4 Dense index with search key *dept_name*.

block. The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block (see Section 13.3.2), we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.

For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

14.2.2 Multilevel Indices

Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4-kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy b blocks, binary search requires as many as $\lceil \log_2(b) \rceil$ blocks to be read. ($\lceil x \rceil$ denotes the least integer that is greater than or equal to x; that is, we round upward.) Note that the blocks that are read are not adjacent

Telegram: @uni_k

www.konkur.in

to each other, so each read requires a random (i.e., non-sequential) I/O operation. For a 10,000-block index, binary search requires 14 random block reads. On a magnetic disk system where a random block read takes on average 10 milliseconds, the index search will take 140 milliseconds. This may not seem much, but we would be able to carry out only seven index searches a second on a single disk, whereas a more efficient search mechanism would let us carry out far more searches per second, as we shall see shortly. Note that, if overflow blocks have been used, binary search is only possible on the non-overflow blocks, and the actual cost may be even higher than the logarithmic bound above. A sequential search requires b sequential block reads, which may take even longer (although in some cases the lower cost of sequential block reads may result in sequential search being faster than a binary search). Thus, the process of searching a large index may be costly.

To deal with this problem, we treat the index just as we would treat any other sequential file, and we construct a sparse outer index on the original index, which we now call the inner index, as shown in Figure 14.5. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less

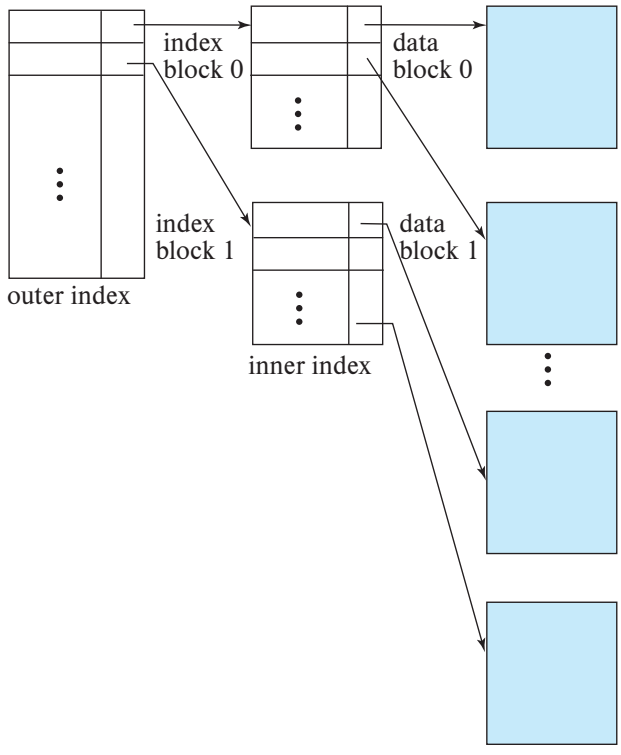


Figure 14.5 Two-level sparse index.

Telegram: @uni_k

www.konkur.in

than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

In our example, an inner index with 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks. If we assume that the outer index is already in main memory, we would read only one index block for a search using a multilevel index, rather than the 14 blocks we read with binary search. As a result, we can perform 14 times as many index searches per second.

If our file is extremely large, even the outer index may grow too large to fit in main memory. With a 100,000,000-tuple relation, the inner index would occupy 1,000,000 blocks, and the outer index would occupy 10,000 blocks, or 40 megabytes. Since there are many demands on main memory, it may not be possible to reserve that much main memory just for this particular outer index. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel indices**. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search.[2]

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing. We shall examine the relationship later, in Section 14.3.

14.2.3 Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. Further, in case a record in the file is updated, any index whose search-key attribute is affected by the update must also be updated; for example, if the department of an instructor is changed, an index on the *dept_name* attribute of *instructor* must be updated correspondingly. Such a record update can be modeled as a deletion of the old record, followed by an insertion of the new value of the record, which results in an index deletion followed by an index insertion. As a result we only need to consider insertion and deletion on an index, and we do not need to consider updates explicitly.

We first describe algorithms for updating single-level indices.

14.2.3.1 Insertion

First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:

[2]In the early days of disk-based indices, each level of the index corresponded to a unit of physical storage. Thus, we may have indices at the track, cylinder, and disk levels. Such a hierarchy does not make sense today since disk subsystems hide the physical details of disk storage, and the number of disks and platters per disk is very small compared to the number of cylinders or bytes per track.

Telegram: @uni_k

www.konkur.in

- Dense indices:
  1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
  2. Otherwise the following actions are taken:
     a. If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry.
     b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
- Sparse indices: We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

14.2.3.2 Deletion

To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:

- Dense indices:
  1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
  2. Otherwise the following actions are taken:
     a. If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry.
     b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.
- Sparse indices:
  1. If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
  2. Otherwise the system takes the following actions:
     a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Telegram: @uni_k

www.konkur.in

b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

Insertion and deletion algorithms for multilevel indices are a simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest-level index is merely a file containing records—thus, if there is any change in the lowest-level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

14.2.4 Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

If a relation can have more than one record containing the same search key value (that is, two or more records can have the same values for the indexed attributes), the search key is said to be a **nonunique search key**.

One way to implement secondary indices on nonunique search keys is as follows: Unlike the case of primary indices, the pointers in such a secondary index do not point directly to the records. Instead, each pointer in the index points to a bucket that in turn contains pointers to the file. Figure 14.6 shows the structure of a secondary index that uses such an extra level of indirection on the *instructor* file, on the search key *dept_name*.

However, this approach has a few drawbacks. First, index access takes longer, due to an extra level of indirection, which may require a random operation. Second,

Telegram: @uni_k

www.konkur.in

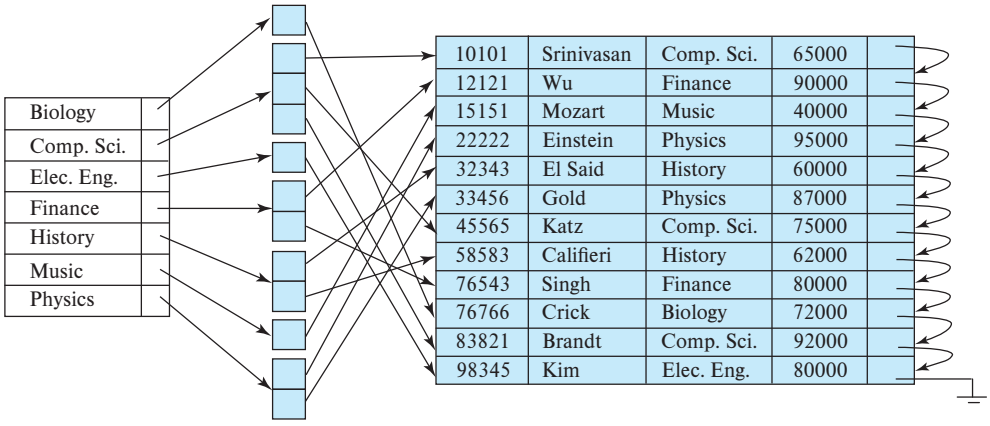


Figure 14.6 Secondary index on *instructor* file, on noncandidate key *dept_name*.

if a key has very few or no duplicates, if a whole block is allocated to its associated bucket, a lot of space would be wasted. Later in this chapter, we study more efficient alternatives for implementing secondary indices, which avoid these drawbacks.

A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index. Because secondary-key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.

The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, *every* index must be updated.

Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

14.2.5 Indices on Multiple Keys

Although the examples we have seen so far have had a single attribute in a search key, in general a search key can have more than one attribute. A search key containing more than one attribute is referred to as a **composite search key**. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form $(a_1, \ldots, a_n)$, where the indexed attributes are A , … , A .

Telegram: @uni_k

www.konkur.in

The ordering of search-key values is the *lexicographic ordering*. For example, for the case of two attribute search keys, $(a_1, a_2) < (b_1, b_2)$ if either $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$. Lexicographic ordering is basically the same as alphabetic ordering of words.

As an example, consider an index on the *takes* relation, on the composite search key (*course_id*, *semester*, *year*). Such an index would be useful to find all students who have registered for a particular course in a particular semester/year. An ordered index on a composite key can also be used to answer several other kinds of queries efficiently, as we shall see in Section 14.6.2.

14.3 B$^+$-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

The **B$^+$-tree index** structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B$^+$-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree (other than the root) has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree; the root has between 2 and n children.

We shall see that the B$^+$-tree structure imposes performance overhead on insertion and deletion and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B$^+$-tree structure.

14.3.1 Structure of a B$^+$-Tree

A B$^+$-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. We assume for now that there are no duplicate search key values, that is, each search key is unique and occurs in at most one record; we consider the issue of nonunique search keys later.

Figure 14.7 shows a typical node of a B$^+$-tree. It contains up to $n - 1$ search-key values $K_1, K_2, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.



Figure 14.7 Typical node of a B$^+$-tree.

Telegram: @uni_k

www.konkur.in

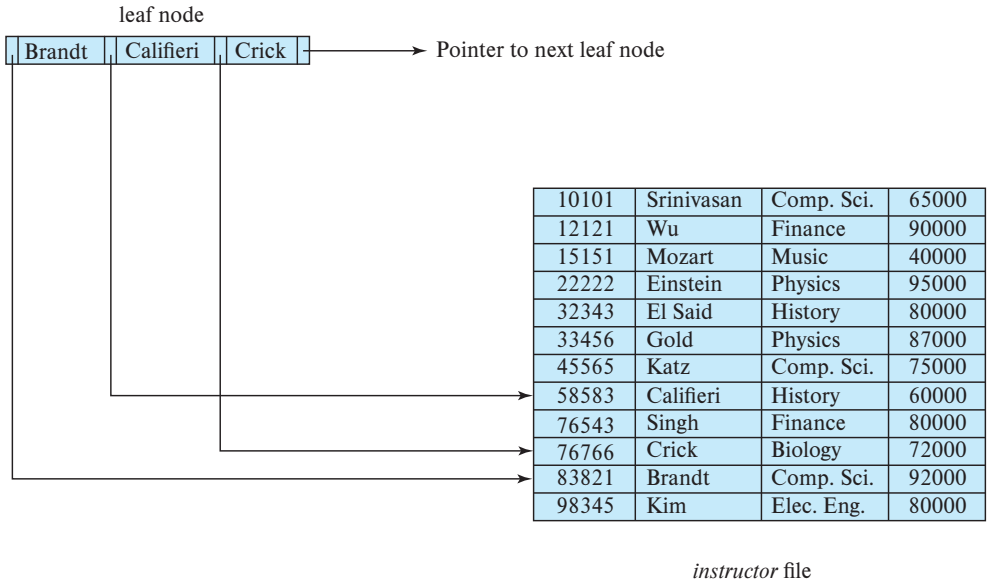


Figure 14.8 A leaf node for *instructor* B+-tree index ($n = 4$).

We consider first the structure of the **leaf nodes**. For $i = 1, 2, \ldots, n - 1$, pointer P_i points to a file record with search-key value K_i. Pointer P_n has a special purpose that we shall discuss shortly.

Figure 14.8 shows one leaf node of a B+-tree for the *instructor* file, in which we have chosen n to be 4, and the search key is *name*.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $\lceil (n - 1)/2 \rceil$ values. With $n = 4$ in our example B+-tree, each leaf must contain at least two values, and at most three values.

If L_i and L_j are leaf nodes and $i < j$ (that is, L_i is to the left of L_j in the tree), then every search-key value v_i in L_i is less than every search-key value v_j in L_j.

If the B+-tree index is used as a dense index (as is usually the case), every search-key value must appear in some leaf node.

Now we can explain the use of the pointer P_n. Since there is a linear order on the leaves based on the search-key values that they contain, we use P_n to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

The **nonleaf nodes** of the B+-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers and *must* hold at least $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fanout* of the node. Nonleaf nodes are also referred to as **internal nodes**.

Telegram: @uni_k

www.konkur.in

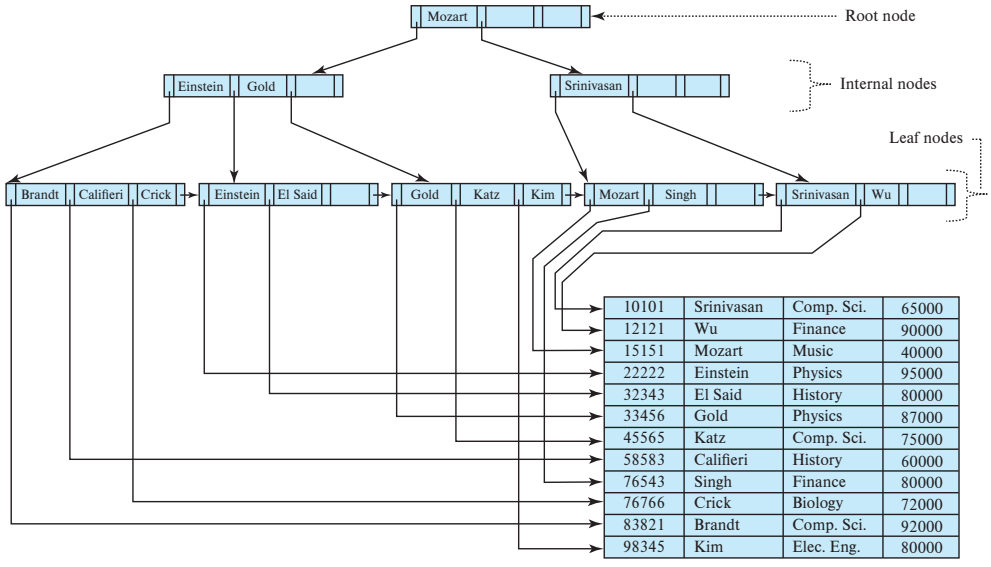


Figure 14.9 B$^+$-tree for *instructor* file ($n = 4$).

Let us consider a node containing m pointers ($m \leq n$). For $i = 2, 3, \ldots, m - 1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}, and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1.

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B$^+$-tree, for any n, that satisfies the preceding requirements.

Figure 14.9 shows a complete B$^+$-tree for the *instructor* file (with $n = 4$). We have omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.

Figure 14.10 shows another B$^+$-tree for the *instructor* file, this time with $n = 6$. Observe that the height of this tree is less than that of the previous tree, which had $n = 4$.

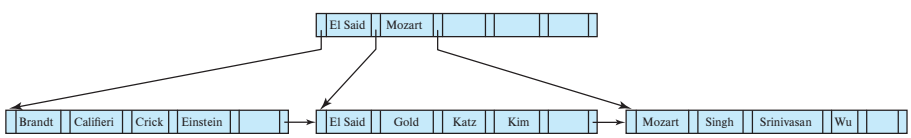


Figure 14.10 B$^+$-tree for *instructor* file with $n = 6$.

Telegram: @uni_k

www.konkur.in

These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B+-tree. Indeed, the "B" in B+-tree stands for "balanced." It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.

In general, search keys could have duplicates. One way to handle the case of nonunique search keys is to modify the tree structure to store each search key at a leaf node as many times as it appears in records, with each copy pointing to one record. The condition that $K_i < K_j$ if $i < j$ will need to be modified to $K_i \leq K_j$. However, this approach can result in duplicate search key values at internal nodes, making the insertion and deletion procedures more complicated and expensive. Another alternative is to store a set (or bucket) of record pointers with each search key value, as we saw earlier. This approach is more complicated and can result in inefficient access, especially if the number of record pointers for a particular key is very large.

Most database implementations instead make search keys unique as follows: Suppose the desired search key attribute a_i of relation r is nonunique. Let A_p be the primary key of r. Then the unique composite search key (a_i, A_p) is used instead of a_i when building the index. (Any set of attributes that together with a_i guarantee uniqueness can also be used instead of A_p.) For example, if we wished to create an index on the *instructor* relation on the attribute *name*, we instead create an index on the composite search key (*name*, ID), since ID is the primary key for *instructor*. Index lookups on just *name* can be efficiently handled using this index, as we shall see shortly. Section 14.3.5 covers issues in handling of nonunique search keys in more detail.

In our examples, we show indices on some nonunique search keys, such as *instructor.name*, assuming for simplicity that there are no duplicates; in reality most databases would automatically add extra attributes internally, to ensure the absence of duplicates.

14.3.2 Queries on B+-Trees

Let us consider how we process queries on a B+-tree. Suppose that we wish to find a record with a given value v for the search key. Figure 14.11 presents pseudocode for a function *find*(v) to carry out this task, assuming there are no duplicates, that is, there is at most one record with a particular search key. We address the issue of nonunique search keys later in this section.

Intuitively, the function starts at the root of the tree and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest i such that search-key value K_i is greater than or equal to v. Suppose such a value is found; then, if K_i is equal to v, the current node is set to the node pointed to by P_{i+1}, otherwise $K_i > v$, and the current node is set to the node pointed to by P_i. If no such value K_i is found, then $v > K_{m-1}$, where P_m is the last nonnull pointer in the node. In this case the current node is set to that pointed to by P_m. The above procedure is repeated, traversing down the tree until a leaf node is reached.

Telegram: @uni_k

www.konkur.in


```
function find(v)
/* Assumes no duplicate keys, and returns pointer to the record with
 * search key value v if such a record exists, and null otherwise */
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that v ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (v = C.K_i) then Set C = C.P_{i+1}
        else Set C = C.P_i /* v < C.K_i */
    end
    /* C is a leaf node */
    if for some i, K_i = v
        then return P_i
        else return null ; /* No record with key value v exists*/
```

Figure 14.11 Querying a B$^+$-tree.

At the leaf node, if there is a search-key value $K_i = v$, pointer P_i directs us to a record with search-key value K_i. The function then returns the pointer to the record, P_i. If no search key with value v is found in the leaf node, no record with key value v exists in the relation, and function *find* returns null, to indicate failure.

B$^+$-trees can also be used to find all records with search key values in a specified range [*lb*, *ub*]. For example, with a B$^+$-tree on attribute *salary* of *instructor*, we can find all *instructor* records with salary in a specified range such as [50000, 100000] (in other words, all salaries between 50000 and 100000). Such queries are called **range queries**.

To execute such queries, we can create a procedure *findRange*(*lb*, *ub*), shown in Figure 14.12. The procedure does the following: it first traverses to a leaf in a manner similar to *find*(*lb*); the leaf may or may not actually contain value *lb*. It then steps through records in that and subsequent leaf nodes collecting pointers to all records with key values $C.K_i$ s.t. $lb \leq C.K_i \leq ub$ into a set resultSet. The function stops when $C.K_i > ub$, or there are no more keys in the tree.

A real implementation would provide a version of *findRange* supporting an iterator interface similar to that provided by the JDBC `ResultSet`, which we saw in Section 5.1.1. Such an iterator interface would provide a method *next*(), which can be called repeatedly to fetch successive records. The *next*() method would step through the entries at the leaf level, in a manner similar to *findRange*, but each call takes only one step and records where it left off, so that successive calls to *next*() step through successive en-

Telegram: @uni_k

www.konkur.in


```
function findRange(lb, ub)
/* Returns all records with search key value V such that lb ≤ V ≤ ub. */
    Set resultSet = {};
    Set C = root node
    while (C is not a leaf node) begin
        Let i = smallest number such that lb ≤ C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m
        end
        else if (lb = C.K_i) then Set C = C.P_{i+1}
        else Set C = C.P_i /* lb < C.K_i */
    end
    /* C is a leaf node */
    Let i be the least value such that K_i ≥ lb
    if there is no such i
        then Set i = 1 + number of keys in C; /* To force move to next leaf */
    Set done = false;
    while (not done) begin
        Let n = number of keys in C.
        if ( i ≤ n and C.K_i ≤ ub) then begin
            Add C.P_i to resultSet
            Set i = i + 1
        end
        else if  (i ≤ n and C.K_i > ub)
            then Set done = true;
        else if (i > n and C.P_{n+1} is not null)
            then Set C = C.P_{n+1}, and i = 1 /* Move to next leaf */
        else Set done = true; /* No more leaves to the right */
    end
    return resultSet;
```

Figure 14.12 Range query on a B+-tree.

tries. We omit details for simplicity, and leave the pseudocode for the iterator interface as an exercise for the interested reader.

We now consider the cost of querying on a B+-tree index. In processing a query, we traverse a path in the tree from the root to some leaf node. If there are N records in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$.

Typically, the node size is chosen to be the same as the size of a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of

Telegram: @uni_k

www.konkur.in

8 bytes, n is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, n is around 100. With $n = 100$, if we have 1 million search-key values in the file, a lookup requires only $\lceil \log_{50}(1{,}000{,}000) \rceil = 4$ nodes to be accessed. Thus, at most four blocks need to be read from disk to traverse the path from the root to a leaf. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.

An important difference between B$^+$-tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small and has at most two pointers. In a B$^+$-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B$^+$-trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length $\lceil \log_2(N) \rceil$, where N is the number of records in the file being indexed. With $N = 1{,}000{,}000$ as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the B$^+$-tree. The difference is significant with a magnetic disk, since each block read could require a disk arm seek which, together with the block read, takes about 10 milliseconds on a magnetic disk. The difference is not quite as drastic with flash storage, where a read of a 4 kilobyte page takes around 10 to 100 microseconds, but it is still significant.

After traversing down to the leaf level, queries on a single value of a unique search key require one more random I/O operation to fetch any matching record.

Range queries have an additional cost, after traversing down to the leaf level: all the pointers in the given range must be retrieved. These pointers are in consecutive leaf nodes; thus, if M such pointers are retrieved, at most $\lceil M/(n/2) \rceil + 1$ leaf nodes need to be accessed to retrieve the pointers (since each leaf node has at least $n/2$ pointers, but even two pointers may be split across two pages). To this cost, we need to add the cost of accessing the actual records. For secondary indices, each such record may be on a different block, which could result in M random I/O operations in the worst case. For clustered indices, these records would be in consecutive blocks, with each block containing multiple records, resulting in a significantly lower cost.

Now, let us consider the case of nonunique keys. As explained earlier, if we wish to create an index on an attribute a_i that is not a candidate key, and may thus have duplicates, we instead create an index on a composite key that is duplicate-free. The composite key is created by adding extra attributes, such as the primary key, to a_i, to ensure uniqueness. Suppose we created an index on the composite key (a_i, A_p) instead of creating an index on a_i.

An important question, then, is how do we retrieve all tuples with a given value v for a_i using the above index? This question is easily answered by using the function findRange(lb, ub), with $lb = (v, -\infty)$ and $ub = (v, \infty)$, where $-\infty$ and ∞ denote the smallest and largest possible values of A_p. All records with $a_i = v$ would be returned by the above function call. Range queries on a_i can be handled similarly. These range

Telegram: @uni_k

www.konkur.in

queries retrieve pointers to the records quite efficiently, although retrieval of the records may be expensive, as discussed earlier.

14.3.3 Updates on B+-Trees

When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly. Recall that updates to a record can be modeled as a deletion of the old record followed by insertion of the updated record. Hence we only consider the case of insertion and deletion.

Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (i.e., combine nodes) if a node becomes too small (fewer than $\lceil n/2 \rceil$ pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and deletion in a B+-tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

- **Insertion**. Using the same technique as for lookup from the *find*() function (Figure 14.11), we first find the leaf node in which the search-key value would appear. We then insert an entry (i.e., a search-key value and record pointer pair) in the leaf node, positioning it such that the search keys are still in order.
- **Deletion**. Using the same technique as for lookup, we find the leaf node containing the entry to be deleted by performing a lookup on the search-key value of the deleted record; if there are multiple entries with the same search-key value, we search across all entries with the same search-key value until we find the entry that points to the record being deleted. We then remove the entry from the leaf node. All entries in the leaf node that are to the right of the deleted entry are shifted left by one position, so that there are no gaps in the entries after the entry is deleted.

We now consider the general case of insertion and deletion, dealing with node splitting and node coalescing.

14.3.3.1 Insertion

We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for "Adams" into the B+-tree of Figure 14.9. Using the algorithm for lookup, we find that "Adams" should appear in the leaf node containing "Brandt", "Califieri", and "Crick." There is no room in this leaf to insert the search-key value "Adams." Therefore, the node is *split* into two nodes. Figure 14.13 shows the two leaf nodes that result from the split of the leaf node on inserting "Adams". The search-key values "Adams" and "Brandt" are in one leaf, and "Califieri" and "Crick" are in the other. In general, we take the n search-key values (the $n - 1$ values in the leaf

Telegram: @uni_k

www.konkur.in

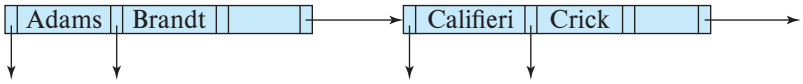


Figure 14.13 Split of leaf node on insertion of "Adams".

node plus the value being inserted), and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a newly created node.

Having split a leaf node, we must insert the new leaf node into the B$^+$-tree structure. In our example, the new node has "Califieri" as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B$^+$-tree of Figure 14.14 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

Splitting of a nonleaf node is a little different from splitting of a leaf node. Figure 14.15 shows the result of inserting a record with search key "Lamport" into the tree shown in Figure 14.14. The leaf node in which "Lamport" is to be inserted already has entries "Gold", "Katz", and "Kim", and as a result the leaf node has to be split. The new right-hand-side node resulting from the split contains the search-key values "Kim" and "Lamport". An entry (Kim, $n1$) must then be added to the parent node, where $n1$ is a pointer to the new node, However, there is no space in the parent node to add a new entry, and the parent node has to be split. To do so, the parent node is conceptually expanded temporarily, the entry added, and the overfull node is then immediately split.

When an overfull nonleaf node is split, the child pointers are divided among the original and the newly created nodes; in our example, the original node is left with the first three pointers, and the newly created node to the right gets the remaining two pointers. The search key values are, however, handled a little differently. The search key values that lie between the pointers moved to the right node (in our example, the value "Kim") are moved along with the pointers, while those that lie between the pointers that stay on the left (in our example, "Califieri" and "Einstein") remain undisturbed.

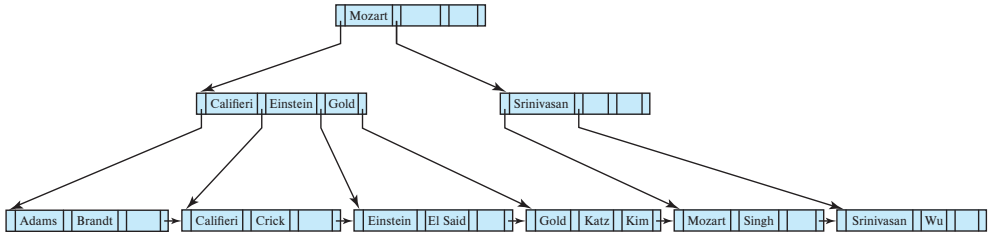


Figure 14.14 Insertion of "Adams" into the B$^+$-tree of Figure 14.9.

Telegram: @uni_k

www.konkur.in

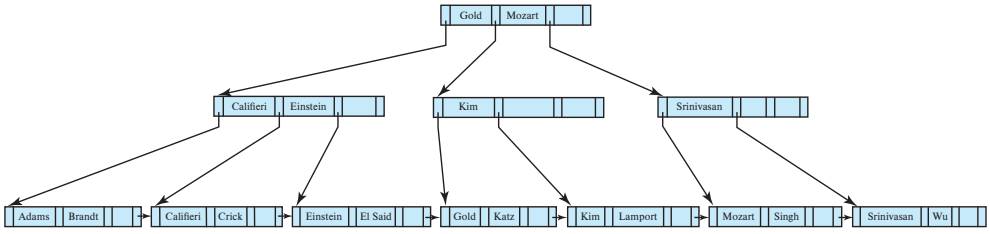


Figure 14.15 Insertion of "Lamport" into the B+-tree of Figure 14.14.

However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. In our example, the search key value "Gold" lies between the three pointers that went to the left node, and the two pointers that went to the right node. The value "Gold" is not added to either of the split nodes. Instead, an entry (Gold, $n2$) is added to the parent node, where $n2$ is a pointer to the newly created node that resulted from the split. In this case, the parent node is the root, and it has enough space for the new entry.

The general technique for insertion into a B+-tree is to determine the leaf node l into which insertion must occur. If a split results, insert the new node into the parent

---

procedure *insert*(*value* K, *pointer* P)
  if (tree is empty) create an empty leaf node L, which is also the root
  else Find the leaf node L that should contain key value K
  if (L has less than $n - 1$ key values)
    then insert_in_leaf (L, K, P)
    else begin /* L has $n - 1$ key values already, split it */
      Create node L'
      Copy $L.P_1 \ldots L.K_{n-1}$ to a block of memory T that can
        hold n (pointer, key-value) pairs
      insert_in_leaf (T, K, P)
      Set $L'.P_n = L.P_n$; Set $L.P_n = L'$
      Erase $L.P_1$ through $L.K_{n-1}$ from L
      Copy $T.P_1$ through $T.K_{\lceil n/2 \rceil}$ from T into L starting at $L.P_1$
      Copy $T.P_{\lceil n/2 \rceil + 1}$ through $T.K_n$ from T into L' starting at $L'.P_1$
      Let K' be the smallest key-value in L'
      insert_in_parent(L, K', L')
    end

Figure 14.16 Insertion of entry in a B+-tree.

Telegram: @uni_k

www.konkur.in

of node l. If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

Figure 14.16 outlines the insertion algorithm in pseudocode. The procedure *insert* inserts a key-value pointer pair into the index, using two subsidiary procedures *insert_in_leaf* and *insert_in_parent*, shown in Figure 14.17. In the pseudocode, L, N, P and T denote pointers to nodes, with L being used to denote a leaf node. $L.K_i$ and $L.P_i$ denote the ith value and the ith pointer in node L, respectively; $T.K_i$ and $T.P_i$ are used similarly. The pseudocode also makes use of the function $parent(N)$ to find the parent of a node N. We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and we can use it later to find the parent of any node in the path efficiently.

```
procedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K_1)
        then insert P, K into L just before L.P_1
        else begin
            Let K_i be the highest value in L that is less than or equal to K
            Insert P, K into L just after L.K_i
        end
procedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'    /* N and N' are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P_1 ... T.P_⌈(n+1)/2⌉ into P
            Let K'' = T.K_⌈(n+1)/2⌉
            Copy T.P_⌈(n+1)/2⌉+1 ... T.P_n+1 into P'
            insert_in_parent(P, K'', P')
        end
```

Figure 14.17 Subsidiary procedures for insertion of entry in a B$^+$-tree.

Telegram: @uni_k

www.konkur.in

The procedure *insert_in_parent* takes as parameters N, K', N', where node N was split into N and N', with K' being the least value in N'. The procedure modifies the parent of N to record the split. The procedures *insert_into_index* and *insert_in_parent* use a temporary area of memory T to store the contents of a node being split. The procedures can be modified to copy data from the node being split directly to the newly created node, reducing the time required for copying data. However, the use of the temporary space T simplifies the procedures.

14.3.3.2 Deletion

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete "Srinivasan" from the B+-tree of Figure 14.14. The resulting B+-tree appears in Figure 14.18. We now consider how the deletion is performed. We first locate the entry for "Srinivasan" by using our lookup algorithm. When we delete the entry for "Srinivasan" from its leaf node, the node is left with only one entry, "Wu". Since, in our example, $n = 4$ and $1 < \lceil (n-1)/2 \rceil$, we must either merge the node with a sibling node or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for "Wu" can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling and deleting the now-empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

In our example, the entry to be deleted is (Srinivasan, $n3$), where $n3$ is a pointer to the leaf containing "Srinivasan". (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.) After deleting the above entry, the parent node, which had a search key value "Srinivasan" and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since $1 < \lceil n/2 \rceil$ for $n = 4$, the parent node is underfull. (For larger n, a node that becomes underfull would still have some values as well as pointers.)

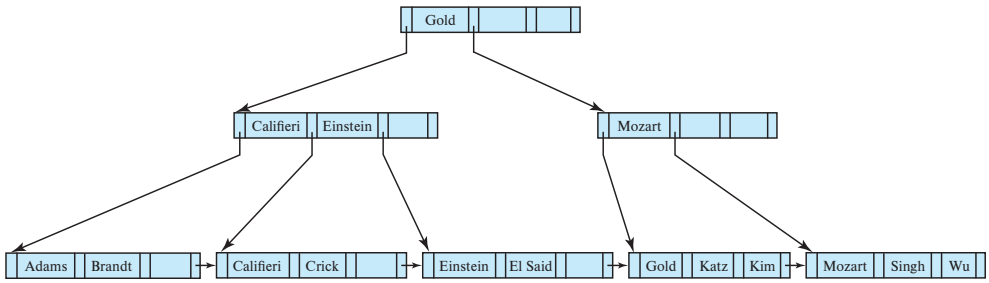


Figure 14.18 Deletion of "Srinivasan" from the B+-tree of Figure 14.14.

Telegram: @uni_k

www.konkur.in

In this case, we look at a sibling node; in our example, the only sibling is the nonleaf node containing the search keys "Califieri", "Einstein", and "Gold". If possible, we try to coalesce the node with its sibling. In this case, coalescing is not possible, since the node and its sibling together have five pointers, against a maximum of four. The solution in this case is to **redistribute** the pointers between the node and its sibling, such that each has at least $\lceil n/2 \rceil = 2$ child pointers. To do so, we move the rightmost pointer from the left sibling (the one pointing to the leaf node containing "Gold") to the underfull right sibling. However, the underfull right sibling would now have two pointers, namely, its leftmost pointer, and the newly moved pointer, with no value separating them. In fact, the value separating them is not present in either of the nodes, but is present in the parent node, between the pointers from the parent to the node and its sibling. In our example, the value "Mozart" separates the two pointers and is present in the right sibling after the redistribution. Redistribution of the pointers also means that the value "Mozart" in the parent no longer correctly separates search-key values in the two siblings. In fact, the value that now correctly separates search-key values in the two sibling nodes is the value "Gold", which was in the left sibling before redistribution.

As a result, as can be seen in the B^+-tree in Figure 14.18, after redistribution of pointers between siblings, the value "Gold" has moved up into the parent, while the value that was there earlier, "Mozart", has moved down into the right sibling.

We next delete the search-key values "Singh" and "Wu" from the B^+-tree of Figure 14.18. The result is shown in Figure 14.19. The deletion of the first of these values does not make the leaf node underfull, but the deletion of the second value does. It is not possible to merge the underfull node with its sibling, so a redistribution of values is carried out, moving the search-key value "Kim" into the node containing "Mozart", resulting in the tree shown in Figure 14.19. The value separating the two siblings has been updated in the parent, from "Mozart" to "Kim".

Now we delete "Gold" from the above tree; the result is shown in Figure 14.20. This results in an underfull leaf, which can now be merged with its sibling. The resultant deletion of an entry from the parent node (the nonleaf node containing "Kim") makes the parent underfull (it is left with just one pointer). This time around, the parent node can be merged with its sibling. This merge results in the search-key value "Gold"

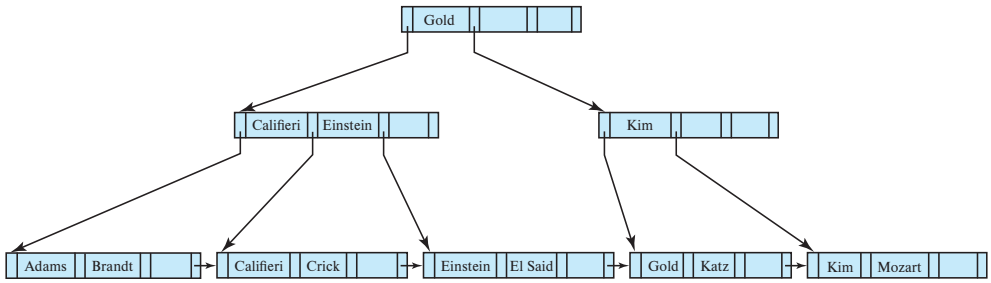


Figure 14.19 Deletion of "Singh" and "Wu" from the B^+-tree of Figure 14.18.

Telegram: @uni_k

www.konkur.in

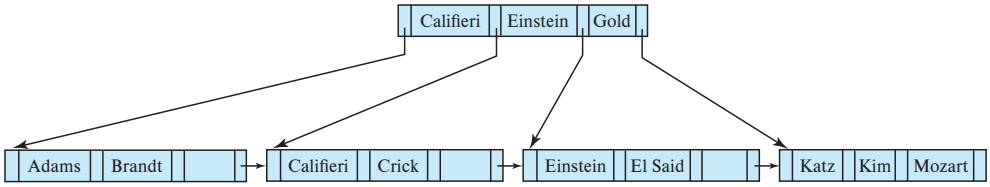


Figure 14.20 Deletion of "Gold" from the B+-tree of Figure 14.19.

moving down from the parent into the merged node. As a result of this merge, an entry is deleted from its parent, which happens to be the root of the tree. And as a result of that deletion, the root is left with only one child pointer and no search-key value, violating the condition that the root must have at least two children. As a result, the root node is deleted and its sole child becomes the root, and the depth of the B+-tree has been decreased by 1.

It is worth noting that, as a result of deletion, a key value that is present in a nonleaf node of the B+-tree may not be present at any leaf of the tree. For example, in Figure 14.20, the value "Gold" has been deleted from the leaf level but is still present in a nonleaf node.

In general, to delete a value in a B+-tree, we perform a lookup on the value and delete it. If the node is too small, we delete it from its parent. This deletion results in recursive application of the deletion algorithm until the root is reached, a parent remains adequately full after deletion, or redistribution is applied.

Figure 14.21 outlines the pseudocode for deletion from a B+-tree. The procedure *swap_variables*(N, N') merely swaps the values of the (pointer) variables N and N'; this swap has no effect on the tree itself. The pseudocode uses the condition "too few pointers/values." For nonleaf nodes, this criterion means less than $\lceil n/2 \rceil$ pointers; for leaf nodes, it means less than $\lceil (n-1)/2 \rceil$ values. The pseudocode redistributes entries by borrowing a single entry from an adjacent node. We can also redistribute entries by repartitioning entries equally between the two nodes. The pseudocode refers to deleting an entry (K, P) from a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the pointer P precedes the key value K. For nonleaf nodes, P follows the key value K.

14.3.4 Complexity of B+-Tree Updates

Although insertion and deletion operations on B+-trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed in the worst case for an insertion is proportional to $\log_{\lceil n/2 \rceil}(N)$, where n is the maximum number of pointers in a node, and N is the number of records in the file being indexed.

The worst-case complexity of the deletion procedure is also proportional to $\log_{\lceil n/2 \rceil}(N)$, provided there are no duplicate values for the search key; we discuss the case of nonunique search keys later in this chapter.

Telegram: @uni_k

www.konkur.in


```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
                        pair in N'
                    remove (N'.P_m, N'.K_m) from N'
                    insert (N'.P_m, N'.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_m
                end
            end
            else … symmetric to the then case …
        end
    end
```

Figure 14.21 Deletion of entry from a B$^+$-tree.

Telegram: @uni_k

www.konkur.in

In other words, the cost of insertion and deletion operations in terms of I/O operations is proportional to the height of the B+-tree, and is therefore low. It is the speed of operation on B+-trees that makes them a frequently used index structure in database implementations.

In practice, operations on B+-trees result in fewer I/O operations than the worst-case bounds. With fanout of 100, and assuming accesses to leaf nodes are uniformly distributed, the parent of a leaf node is 100 times more likely to get accessed than the leaf node. Conversely, with the same fanout, the total number of nonleaf nodes in a B+-tree would be just a little more than 1/100th of the number of leaf nodes. As a result, with memory sizes of several gigabytes being common today, for B+-trees that are used frequently, even if the relation is very large it is quite likely that most of the nonleaf nodes are already in the database buffer when they are accessed. Thus, typically only one or two I/O operations are required to perform a lookup. For updates, the probability of a node split occurring is correspondingly very small. Depending on the ordering of inserts, with a fanout of 100, only from 1 in 100 to 1 in 50 insertions will result in a node split, requiring more than one block to be written. As a result, on an average an insert will require just a little more than one I/O operation to write updated blocks.

Although B+-trees only guarantee that nodes will be at least half full, if entries are inserted in random order, nodes can be expected to be more than two-thirds full on average. If entries are inserted in sorted order, on the other hand, nodes will be only half full. (We leave it as an exercise to the reader to figure out why nodes would be only half full in the latter case.)

14.3.5 Nonunique Search Keys

We have assumed so far that search keys are unique. Recall also that we described earlier, in Section 14.3.1, how to make search keys unique by creating a composite search key containing the original search key and extra attributes, that together are unique across all records.

The extra attribute can be a record-id, which is a pointer to the record, or a primary key, or any other attribute whose value is unique among all records with the same search-key value. The extra attribute is called a **uniquifier** attribute.

A search with the original search-key attribute can be carried out using a range search as we saw in Section 14.3.2; alternatively, we can create a variant of the *findRange* function that takes only the original search key value as parameter and ignores the value of the uniquifier attribute when comparing search-key values.

It is also possible to modify the B+-tree structure to support duplicate search keys. The insert, delete, and lookup methods all have to be modified correspondingly.

- One alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys. This approach is space efficient since it stores the key value only once; however, it creates several complications when B+-trees are implemented. If the

Telegram: @uni_k

www.konkur.in

buckets are kept in the leaf node, extra code is needed to deal with variable-size buckets, and to deal with buckets that grow larger than the size of the leaf node. If the buckets are stored in separate blocks, an extra I/O operation may be required to fetch records.

- Another option is to store the search key value once per record; this approach allows a leaf node to be split in the usual way if it is found to be full during an insert. However, this approach makes handling of split and search on internal nodes significantly more complicated, since two leaves may contain the same search key value. It also has a higher space overhead, since key values are stored as many times as there are records containing that value.

A major problem with both these approaches, as compared to the unique search-key approach, lies in the efficiency of record deletion. (The complexity of lookup and insertion are the same with both these approaches, as well as with the unique search-key approach.) Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to search through a number of entries with the same search-key value, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted. Thus, the worst-case complexity of deletion may be linear in the number of records.

In contrast, record deletion can be done efficiently using the unique search key approach. When a record is to be deleted, the composite search-key value is computed from the record and then used to look up the index. Since the value is unique, the corresponding leaf-level entry can be found with a single traversal from root to leaf, with no further accesses at the leaf level. The worst-case cost of deletion is logarithmic in the number of records, as we saw earlier.

Due to the inefficiency of deletion, as well as other complications due to duplicate search keys, B^+-tree implementations in most database systems only handle unique search keys, and they automatically add record-ids or other attributes to make nonunique search keys unique.

14.4 B^+-Tree Extensions

In this section, we discuss several extensions and variations of the B^+-tree index structure.

14.4.1 B^+-Tree File Organization

As mentioned in Section 14.3, the main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index entries and actual records become out of order and are stored in overflow blocks. We solve the degradation of index lookups by using B^+-tree indices on the file.

Telegram: @uni_k

www.konkur.in

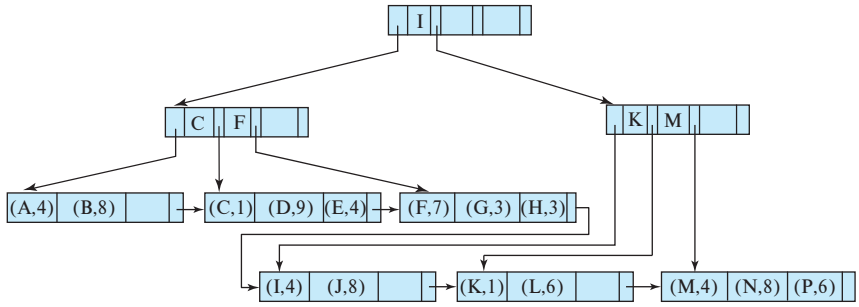


Figure 14.22 B+-tree file organization.

We solve the degradation problem for storing the actual records by using the leaf level of the B+-tree to organize the blocks containing the actual records. We use the B+-tree structure not only as an index, but also as an organizer for records in a file. In a **B+-tree file organization**, the leaf nodes of the tree store records, instead of storing pointers to records. Figure 14.22 shows an example of a B+-tree file organization. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node. However, the leaf nodes are still required to be at least half full.

Insertion and deletion of records from a B+-tree file organization are handled in the same way as insertion and deletion of entries in a B+-tree index. When a record with a given key value v is inserted, the system locates the block that should contain the record by searching the B+-tree for the largest key in the tree that is $\leq v$. If the block located has enough free space for the record, the system stores the record in the block. Otherwise, as in B+-tree insertion, the system splits the block in two and redistributes the records in it (in the B+-tree–key order) to create space for the new record. The split propagates up the B+-tree in the normal fashion. When we delete a record, the system first removes it from the block containing it. If a block B becomes less than half full as a result, the records in B are redistributed with the records in an adjacent block B'. Assuming fixed-sized records, each block will hold at least one-half as many records as the maximum that it can hold. The system updates the nonleaf nodes of the B+-tree in the usual fashion.

When we use a B+-tree for file organization, space utilization is particularly important, since the space occupied by the records is likely to be much more than the space occupied by keys and pointers. We can improve the utilization of space in a B+-tree by involving more sibling nodes in redistribution during splits and merges. The technique is applicable to both leaf nodes and nonleaf nodes, and it works as follows:

During insertion, if a node is full, the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry. If this attempt fails because the adjacent nodes are themselves full, the system splits the node and divides the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node. Since the three nodes together contain one more record

Telegram: @uni_k

www.konkur.in

than can fit in two nodes, each node will be about two-thirds full. More precisely, each node will have at least $\lfloor 2n/3 \rfloor$ entries, where n is the maximum number of entries that the node can hold. ($\lfloor x \rfloor$ denotes the greatest integer that is less than or equal to x; that is, we drop the fractional part, if any.)

During deletion of a record, if the occupancy of a node falls below $\lfloor 2n/3 \rfloor$, the system attempts to borrow an entry from one of the sibling nodes. If both sibling nodes have $\lfloor 2n/3 \rfloor$ records, instead of borrowing an entry, the system redistributes the entries in the node and in the two siblings evenly between two of the nodes and deletes the third node. We can use this approach because the total number of entries is $3\lfloor 2n/3 \rfloor - 1$, which is less than $2n$. With three adjacent nodes used for redistribution, each node can be guaranteed to have $\lfloor 3n/4 \rfloor$ entries. In general, if m nodes ($m-1$ siblings) are involved in redistribution, each node can be guaranteed to contain at least $\lfloor (m-1)n/m \rfloor$ entries. However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

Note that in a B^+-tree index or file organization, leaf nodes that are adjacent to each other in the tree may be located at different places on disk. When a file organization is newly created on a set of records, it is possible to allocate blocks that are mostly contiguous on disk to leaf nodes that are contiguous in the tree. Thus, a sequential scan of leaf nodes would correspond to a mostly sequential scan on disk. As insertions and deletions occur on the tree, sequentiality is increasingly lost, and sequential access has to wait for disk seeks increasingly often. An index rebuild may be required to restore sequentiality.

B^+-tree file organizations can also be used to store large objects, such as SQL clobs and blobs, which may be larger than a disk block, and as large as multiple gigabytes. Such large objects can be stored by splitting them into sequences of smaller records that are organized in a B^+-tree file organization. The records can be sequentially numbered, or numbered by the byte offset of the record within the large object, and the record number can be used as the search key.

14.4.2 Secondary Indices and Record Relocation

Some file organizations, such as the B^+-tree file organization, may change the location of records even when the records have not been updated. As an example, when a leaf node is split in a B^+-tree file organization, a number of records are moved to a new node. In such cases, all secondary indices that store pointers to the relocated records would have to be updated, even though the values in the records may not have changed. Each leaf node may contain a fairly large number of records, and each of them may be in different locations on each secondary index. Thus, a leaf-node split may require tens or even hundreds of I/O operations to update all affected secondary indices, making it a very expensive operation.

A widely used solution for this problem is as follows: In secondary indices, in place of pointers to the indexed records, we store the values of the primary-index search-key

Telegram: @uni_k

www.konkur.in

attributes. For example, suppose we have a primary index on the attribute *ID* of relation *instructor*; then a secondary index on *dept_name* would store with each department name a list of instructor's *ID* values of the corresponding records, instead of storing pointers to the records.

Relocation of records because of leaf-node splits then does not require any update on any such secondary index. However, locating a record using the secondary index now requires two steps: First we use the secondary index to find the primary-index search-key values, and then we use the primary index to find the corresponding records.

This approach thus greatly reduces the cost of index update due to file reorganization, although it increases the cost of accessing data using a secondary index.

14.4.3 Indexing Strings

Creating B^+-tree indices on string-valued attributes raises two problems. The first problem is that strings can be of variable length. The second problem is that strings can be long, leading to a low fanout and a correspondingly increased tree height.

With variable-length search keys, different nodes can have different fanouts even if they are full. A node must then be split if it is full, that is, there is no space to add a new entry, regardless of how many search entries it has. Similarly, nodes can be merged or entries redistributed depending on what fraction of the space in the nodes is used, instead of being based on the maximum number of entries that the node can hold.

The fanout of nodes can be increased by using a technique called **prefix compression**. With prefix compression, we do not store the entire search key value at nonleaf nodes. We only store a prefix of each search key value that is sufficient to distinguish between the key values in the subtrees that it separates. For example, if we had an index on names, the key value at a nonleaf node could be a prefix of a name; it may suffice to store "Silb" at a nonleaf node, instead of the full "Silberschatz" if the closest values in the two subtrees that it separates are, say, "Silas" and "Silver" respectively.

14.4.4 Bulk Loading of B^+-Tree Indices

As we saw earlier, insertion of a record in a B^+-tree requires a number of I/O operations that in the worst case is proportional to the height of the tree, which is usually fairly small (typically five or less, even for large relations).

Now consider the case where a B^+-tree is being built on a large relation. Suppose the relation is significantly larger than main memory, and we are constructing a nonclustering index on the relation such that the index is also larger than main memory. In this case, as we scan the relation and add entries to the B^+-tree, it is quite likely that each leaf node accessed is not in the database buffer when it is accessed, since there is no particular ordering of the entries. With such randomly ordered accesses to blocks, each time an entry is added to the leaf, a disk seek will be required to fetch the block containing the leaf node. The block will probably be evicted from the disk buffer before another entry is added to the block, leading to another disk seek to write the block back

Telegram: @uni_k

www.konkur.in

to disk. Thus, a random read and a random write operation may be required for each entry inserted.

For example, if the relation has 100 million records, and each I/O operation takes about 10 milliseconds on a magnetic disk, it would take at least 1 million seconds to build the index, counting only the cost of reading leaf nodes, not even counting the cost of writing the updated nodes back to disk. This is clearly a very large amount of time; in contrast, if each record occupies 100 bytes, and the disk subsystem can transfer data at 50 megabytes per second, it would take just 200 seconds to read the entire relation.

Insertion of a large number of entries at a time into an index is referred to as **bulk loading** of the index. An efficient way to perform bulk loading of an index is as follows: First, create a temporary file containing index entries for the relation, then sort the file on the search key of the index being constructed, and finally scan the sorted file and insert the entries into the index. There are efficient algorithms for sorting large relations, described later in Section 15.4, which can sort even a large file with an I/O cost comparable to that of reading the file a few times, assuming a reasonable amount of main memory is available.

There is a significant benefit to sorting the entries before inserting them into the B^+-tree. When the entries are inserted in sorted order, all entries that go to a particular leaf node will appear consecutively, and the leaf needs to be written out only once; nodes will never have to be read from disk during bulk load, if the B^+-tree was empty to start with. Each leaf node will thus incur only one I/O operation even though many entries may be inserted into the node. If each leaf contains 100 entries, the leaf level will contain 1 million nodes, resulting in only 1 million I/O operations for creating the leaf level. Even these I/O operations can be expected to be sequential, if successive leaf nodes are allocated on successive disk blocks, and few disk seeks would be required. With magnetic disks, 1 millisecond per block is a reasonable estimate for mostly sequential I/O operations, in contrast to 10 milliseconds per block for random I/O operations.

We shall study the cost of sorting a large relation later, in Section 15.4, but as a rough estimate, the index which would have otherwise taken up to 1,000,000 seconds to build on a magnetic disk can be constructed in well under 1000 seconds by sorting the entries before inserting them into the B^+-tree.

If the B^+-tree is initially empty, it can be constructed faster by building it bottom-up, from the leaf level, instead of using the usual insert procedure. In **bottom-up B^+-tree construction**, after sorting the entries as we just described, we break up the sorted entries into blocks, keeping as many entries in a block as can fit in the block; the resulting blocks form the leaf level of the B^+-tree. The minimum value in each block, along with the pointer to the block, is used to create entries in the next level of the B^+-tree, pointing to the leaf blocks. Each further level of the tree is similarly constructed using the minimum values associated with each node one level below, until the root is created. We leave details as an exercise for the reader.

Most database systems implement efficient techniques based on sorting of entries, and bottom-up construction, when creating an index on a relation, although they use

Telegram: @uni_k

www.konkur.in

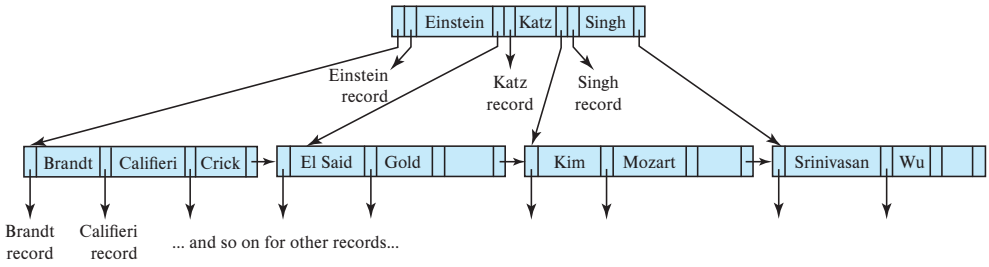


Figure 14.23 B-tree equivalent of B+-tree in Figure 14.9.

the normal insertion procedure when tuples are added one at a time to a relation with an existing index. Some database systems recommend that if a very large number of tuples are added at once to an already existing relation, indices on the relation (other than any index on the primary key) should be dropped, and then re-created after the tuples are inserted, to take advantage of efficient bulk-loading techniques.

14.4.5 B-Tree Index Files

B-tree indices are similar to B+-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the B+-tree of Figure 14.9, the search keys "Einstein", "Gold", "Mozart", and "Srinivasan" appear in nonleaf nodes, in addition to appearing in the leaf nodes. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.

A B-tree allows search-key values to appear only once (if they are unique), unlike a B+-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node. Figure 14.23 shows a B-tree that represents the same search keys as the B+-tree of Figure 14.9. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B+-tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.

It is worth noting that many database system manuals, articles in industry literature, and industry professionals use the term B-tree to refer to the data structure that we call the B+-tree. In fact, it would be fair to say that in current usage, the term B-tree is assumed to be synonymous with B+-tree. However, in this book we use the terms B-tree and B+-tree as they were originally defined, to avoid confusion between the two data structures.

A generalized B-tree leaf node appears in Figure 14.24a; a nonleaf node appears in Figure 14.24b. Leaf nodes are the same as in B+-trees. In nonleaf nodes, the pointers P_i are the tree pointers that we used also for B+-trees, while the pointers B_i are bucket or file-record pointers. In the generalized B-tree in the figure, there are $n - 1$ keys in

Telegram: @uni_k

www.konkur.in

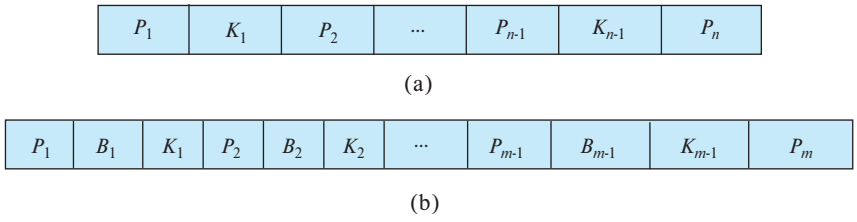


Figure 14.24 Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

the leaf node, but there are $m - 1$ keys in the nonleaf node. This discrepancy occurs because nonleaf nodes must include pointers B_i, thus reducing the number of search keys that can be held in these nodes. Clearly, $m < n$, but the exact relationship between m and n depends on the relative size of search keys and pointers.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B$^+$-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node. However, roughly n times as many keys are stored in the leaf level of a B-tree as in the nonleaf levels, and, since n is typically large, the benefit of finding certain values early is relatively small. Moreover, the fact that fewer search keys appear in a nonleaf B-tree node, compared to B$^+$-trees, implies that a B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B$^+$-tree. Thus, lookup in a B-tree is faster for some search keys but slower for others, although, in general, lookup time is still proportional to the logarithm of the number of search keys.

Deletion in a B-tree is more complicated. In a B$^+$-tree, the deleted entry always appears in a leaf. In a B-tree, the deleted entry may appear in a nonleaf node. The proper value must be selected as a replacement from the subtree of the node containing the deleted entry. Specifically, if search key K_i is deleted, the smallest search key appearing in the subtree of pointer P_{i+1} must be moved to the field formerly occupied by K_i. Further actions need to be taken if the leaf node now has too few entries. In contrast, insertion in a B-tree is only slightly more complicated than is insertion in a B$^+$-tree.

The space advantages of B-trees are marginal for large indices and usually do not outweigh the disadvantages that we have noted. Thus, pretty much all database-system implementations use the B$^+$-tree data structure, even if (as we discussed earlier) they refer to the data structure as a B-tree.

14.4.6 Indexing on Flash Storage

In our description of indexing so far, we have assumed that data are resident on magnetic disks. Although this assumption continues to be true for the most part, flash storage capacities have grown significantly, and the cost of flash storage per gigabyte has dropped correspondingly, and flash based SSD storage has now replaced magnetic-disk storage for many applications.

Telegram: @uni_k

www.konkur.in

Standard B+-tree indices can continue to be used even on SSDs, with acceptable update performance and significantly improved lookup performance compared to disk storage.

Flash storage is structured as pages, and the B+-tree index structure can be used with flash based SSDs. SSDs provide much faster random I/O operations than magnetic disks, requiring only around 20 to 100 microseconds for a random page read, instead of about 5 to 10 milliseconds with magnetic disks. Thus, lookups run much faster with data on SSDs, compared to data on magnetic disks.

The performance of write operations is more complicated with flash storage. An important difference between flash storage and magnetic disks is that flash storage does not permit in-place updates to data at the physical level, although it appears to do so logically. Every update turns into a copy+write of an entire flash-storage page, requiring the old copy of the page to be erased subsequently. A new page can be written in 20 to 100 microseconds, but eventually old pages need to be erased to free up the pages for further writes. Erases are done at the level of blocks containing multiple pages, and a block erase takes 2 to 5 milliseconds.

The optimum B+-tree node size for flash storage is smaller than that with magnetic disk, since flash pages are smaller than disk blocks; it makes sense for tree-node sizes to match to flash pages, since larger nodes would lead to multiple page writes when a node is updated. Although smaller pages lead to taller trees and more I/O operations to access data, random page reads are so much faster with flash storage that the overall impact on read performance is quite small.

Although random I/O is much cheaper with SSDs than with magnetic disks, bulk loading still provides significant performance benefits, compared to tuple-at-a-time insertion, with SSDs. In particular, bottom-up construction reduces the number of page writes compared to tuple-at-a-time insertion, even if the entries are sorted on the search key. Since page writes on flash cannot be done in place and require relatively expensive block erases at a later point in time, the reduction of number of page writes with bottom-up B+-tree construction provides significant performance benefits.

Several extensions and alternatives to B+-trees have been proposed for flash storage, with a focus on reducing the number of erase operations that result due to page rewrites. One approach is to add buffers to internal nodes of B+-trees and record updates temporarily in buffers at higher levels, pushing the updates down to lower levels lazily. The key idea is that when a page is updated, multiple updates are applied together, reducing the number of page writes per update. Another approach creates multiple trees and merges them; the log-structured merge tree and its variants are based on this idea. In fact, both these approaches are also useful for reducing the cost of writes on magnetic disks; we outline both these approaches in Section 14.8.

14.4.7 Indexing in Main Memory

Main memory today is large and cheap enough that many organizations can afford to buy enough main memory to fit all their operational data in-memory. B+-trees can

Telegram: @uni_k

www.konkur.in

be used to index in-memory data, with no change to the structure. However, some optimizations are possible.

First, since memory is costlier than disk space, internal data structures in main memory databases have to be designed to reduce space requirements. Techniques that we saw in Section 14.4.1 to improve B^+-tree storage utilization can be used to reduce memory usage for in-memory B^+-trees.

Data structures that require traversal of multiple pointers are acceptable for in-memory data, unlike in the case of disk-based data, where the cost of the I/Os to traverse multiple pages would be excessively high. Thus, tree structures in main memory databases can be relatively deep, unlike B^+-trees.

The speed difference between cache memory and main memory, and the fact that data are transferred between main memory and cache in units of a *cache-line* (typically about 64 bytes), results in a situation where the relationship between cache and main memory is not dissimilar to the relationship between main memory and disk (although with smaller speed differences). When reading a memory location, if it is present in cache the CPU can complete the read in 1 or 2 nanoseconds, whereas a cache miss results in about 50 to 100 nanoseconds of delay to read data from main memory.

B^+-trees with small nodes that fit in a cache line have been found to provide very good performance with in-memory data. Such B^+-trees allow index operations to be completed with far fewer cache misses than tall, skinny tree structures such as binary trees, since each node traversal is likely to result in a cache miss. Compared to B^+-trees with nodes that match cache lines, trees with large nodes also tend to have more cache misses since locating data within a node requires either a full scan of the node content, spanning multiple cache lines, or a binary search, which also results in multiple cache misses.

For databases where data do not fit entirely in memory, but frequently used data are often memory resident, the following idea is used to create B^+-tree structures that offer good performance on disk as well as in-memory. Large nodes are used to optimize disk-based access, but instead of treating data in a node as single large array of keys and pointers, the data within a node are structured as a tree, with smaller nodes that match the size of a cache line. Instead of scanning data linearly or using binary search within a node, the tree-structure within the large B^+-tree node is used to access the data with a minimal number of cache misses.

14.5 Hash Indices

Hashing is a widely used technique for building indices in main memory; such indices may be transiently created to process a join operation (as we will see in Section 15.5.5) or may be a permanent structure in a main memory database. Hashing has also been used as a way of organizing records in a file, although hash file organizations are not very widely used. We initially consider only in-memory hash indices, and we consider disk-based hashing later in this section.

Telegram: @uni_k

www.konkur.in

In our description of hashing, we shall use the term **bucket** to denote a unit of storage that can store one or more records. For in-memory hash indices, a bucket could be a linked list of index entries or records. For disk-based indices, a bucket would be a linked list of disk blocks. In a **hash file organization**, instead of record pointers, buckets store the actual records; such structures only make sense with disk-resident data. The rest of our description does not depend on whether the buckets store record pointers or actual records.

Formally, let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A **hash function** h is a function from K to B. Let h denote a hash function. With in-memory hash indices, the set of buckets is simply an array of pointers, with the ith bucket at offset i. Each pointer stores the head of a linked list containing the entries in that bucket.

To insert a record with search key K_i, we compute $h(K_i)$, which gives the address of the bucket for that record. We add the index entry for the record to the list at offset i. Note that there are other variants of hash indices that handle the case of multiple records in a bucket differently; the form described here is the most widely used variant and is called **overflow chaining**.

Hash indexing using overflow chaining is also called **closed addressing** (or, less commonly, **closed hashing**). An alternative hashing scheme called open addressing is used in some applications, but is not suitable for most database indexing applications since open addressing does not support deletes efficiently. We do not consider it further.

Hash indices efficiently support equality queries on search keys. To perform a lookup on a search-key value K_i, we simply compute $h(K_i)$, then search the bucket with that address. Suppose that two search keys, K_5 and K_7, have the same hash value; that is, $h(K_5) = h(K_7)$. If we perform a lookup on K_5, the bucket $h(K_5)$ contains records with search-key values K_5 and records with search-key values K_7. Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

Unlike B^+-tree indices, hash indices do not support range queries; for example, a query that wishes to retrieve all search key values v such that $l \leq v \leq u$ cannot be efficiently answered using a hash index.

Deletion is equally straightforward. If the search-key value of the record to be deleted is K_i, we compute $h(K_i)$, then search the corresponding bucket for that record and delete the record from the bucket. With a linked list representation, deletion from the linked list is straightforward.

In a disk-based hash index, when we insert a record, we locate the bucket by using hashing on the search key, as described earlier. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket. If the bucket does not have enough space, a **bucket overflow** is said to occur. We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket b, and b is already full, the system provides an overflow bucket for b and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked

Telegram: @uni_k

www.konkur.in

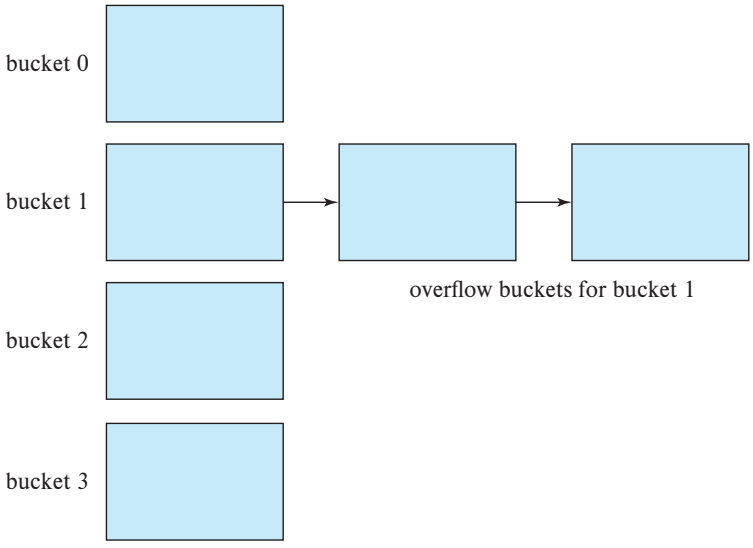


Figure 14.25 Overflow chaining in a disk-based hash structure.

list, as in Figure 14.25. With overflow chaining, given search key k, the lookup algorithm must then search not only bucket $h(k)$, but also the overflow buckets linked from bucket $h(k)$.

Bucket overflow can occur if there are insufficient buckets for the given number of records. If the number of records that are indexed is known ahead of time, the required number of buckets can be allocated; we will shortly see how to deal with situations where the number of records becomes significantly more than what was initially anticipated. Bucket overflow can also occur if some buckets are assigned more records than are others, resulting in one bucket overflowing even when other buckets still have a lot of free space.

Such **skew** in the distribution of records can occur if multiple records may have the same search key. But even if there is only one record per search key, skew may occur if the chosen hash function results in nonuniform distribution of search keys. This chance of this problem can be minimized by choosing hash functions carefully, to ensure the distribution of keys across buckets is uniform and random. Nevertheless, some skew may occur.

So that the probability of bucket overflow is reduced, the number of buckets is chosen to be $(n_r/f_r) * (1 + d)$, where n_r denotes the number of records, f_r denotes the number of records per bucket, d is a fudge factor, typically around 0.2. With a fudge factor of 0.2, about 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

Despite allocation of a few more buckets than required, bucket overflow can still occur, especially if the number of records increases beyond what was initially expected.

Telegram: @uni_k

www.konkur.in

Hash indexing as described above, where the number of buckets is fixed when the index is created, is called **static hashing**. One of the problems with static hashing is that we need to know how many records are going to be stored in the index. If over time a large number of records are added, resulting in far more records than buckets, lookups would have to search through a large number of records stored in a single bucket, or in one or more overflow buckets, and would thus become inefficient.

To handle this problem, the hash index can be rebuilt with an increased number of buckets. For example, if the number of records becomes twice the number of buckets, the index can be rebuilt with twice as many buckets as before. However, rebuilding the index has the drawback that it can take a long time if the relations are large, causing disruption of normal processing. Several schemes have been proposed that allow the number of buckets to be increased in a more incremental fashion. Such schemes are called **dynamic hashing** techniques; the *linear hashing* technique and the *extendable hashing* technique are two such schemes; see Section 24.5 for further details of these techniques.

14.6 Multiple-Key Access

Until now, we have assumed implicitly that only one index on one attribute is used to process a query on a relation. However, for certain types of queries, it is advantageous to use multiple indices if they exist, or to use an index built on a multiattribute search key.

14.6.1 Using Multiple Single-Key Indices

Assume that the *instructor* file has two indices: one for *dept_name* and one for *salary*. Consider the following query: "Find all instructors in the Finance department with salary equal to $80,000." We write

```
select ID
from instructor
where dept_name = 'Finance' and salary = 80000;
```

There are three strategies possible for processing this query:

1. Use the index on *dept_name* to find all records pertaining to the Finance department. Examine each such record to see whether *salary* = 80000.
2. Use the index on *salary* to find all records pertaining to instructors with salary of $80,000. Examine each such record to see whether the department name is "Finance".

Telegram: @uni_k

www.konkur.in

3. Use the index on *dept_name* to find *pointers* to all records pertaining to the Finance department. Also, use the index on *salary* to find pointers to all records pertaining to instructors with a salary of $80,000. Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to instructors of the Finance department and with salary of $80,000.

The third strategy is the only one of the three that takes advantage of the existence of multiple indices. However, even this strategy may be a poor choice if all of the following hold:

- There are many records pertaining to the Finance department.
- There are many records pertaining to instructors with a salary of $80,000.
- There are only a few records pertaining to *both* the Finance department and instructors with a salary of $80,000.

If these conditions hold, we must scan a large number of pointers to produce a small result. An index structure called a "bitmap index" can in some cases greatly speed up the intersection operation used in the third strategy. Bitmap indices are outlined in Section 14.9.

14.6.2 Indices on Multiple Keys

An alternative strategy for this case is to create and use an index on a composite search key (*dept_name*, *salary*)—that is, the search key consisting of the department name concatenated with the instructor salary.

We can use an ordered (B^+-tree) index on the preceding composite search key to answer efficiently queries of the form

```
select ID
from instructor
where dept_name = 'Finance' and salary = 80000;
```

Queries such as the following query, which specifies an equality condition on the first attribute of the search key (*dept_name*) and a range on the second attribute of the search key (*salary*), can also be handled efficiently since they correspond to a range query on the search attribute.

```
select ID
from instructor
where dept_name = 'Finance' and salary < 80000;
```

We can even use an ordered index on the search key (*dept_name*, *salary*) to answer the following query on only one attribute efficiently:

Telegram: @uni_k

www.konkur.in


```
select ID
from instructor
where dept_name = 'Finance';
```

An equality condition *dept_name* = "Finance" is equivalent to a range query on the range with lower end (Finance, $-\infty$) and upper end (Finance, $+\infty$). Range queries on just the *dept_name* attribute can be handled in a similar manner.

The use of an ordered-index structure on a composite search key, however, has a few shortcomings. As an illustration, consider the query

```
select ID
from instructor
where dept_name < 'Finance' and salary < 80000;
```

We can answer this query by using an ordered index on the search key (*dept_name*, *salary*): For each value of *dept_name* that is less than "Finance" in alphabetic order, the system locates records with a *salary* value of 80000. However, each record is likely to be in a different disk block, because of the ordering of records in the file, leading to many I/O operations.

The difference between this query and the previous two queries is that the condition on the first attribute (*dept_name*) is a comparison condition, rather than an equality condition. The condition does not correspond to a range query on the search key.

To speed the processing of general composite search-key queries (which can involve one or more comparison operations), we can use several special structures. We shall consider *bitmap indices* in Section 14.9. There is another structure, called the *R-tree*, that can be used for this purpose. The R-tree is an extension of the B^+-tree to handle indexing on multiple dimensions and is discussed in Section 14.10.1.

14.6.3 Covering Indices

Covering indices are indices that store the values of some attributes (other than the search-key attributes) along with the pointers to the record. Storing extra attribute values is useful with secondary indices, since they allow us to answer some queries using just the index, without even looking up the actual records.

For example, suppose that we have a nonclustering index on the *ID* attribute of the *instructor* relation. If we store the value of the *salary* attribute along with the record pointer, we can answer queries that require the salary (but not the other attribute, *dept_name*) without accessing the *instructor* record.

The same effect could be obtained by creating an index on the search key (*ID*, *salary*), but a covering index reduces the size of the search key, allowing a larger fanout in the nonleaf nodes, and potentially reducing the height of the index.

Telegram: @uni_k

www.konkur.in

14.7 Creation of Indices

Although the SQL standard does not specify any specific syntax for creation of indices, most databases support SQL commands to create and drop indices. As we saw in Section 4.6, indices can be created using the following syntax, which is supported by most databases.

create index <index-name> **on** <relation-name> (<attribute-list>);

The *attribute-list* is the list of attributes of the relations that form the search key for the index. Indices can be dropped using a command of the form

drop index <index-name>;

For example, to define an index named *dept_index* on the *instructor* relation with *dept_name* as the search key, we write:

create index *dept_index* **on** *instructor* (*dept_name*);

To declare that an attribute or list of attributes is a candidate key, we can use the syntax **create unique index** in place of **create index** above. Databases that support multiple types of indices also allow the type of index to be specified as part of the index creation command. Refer to the manual of your database system to find out what index types are available, and the syntax for specifying the index type.

When a user submits an SQL query that can benefit from using an index, the SQL query processor automatically uses the index.

Indices can be very useful on attributes that participate in selection conditions or join conditions of queries, since they can reduce the cost of queries significantly. Consider a query that retrieves *takes* records for a particular student ID 12345 (expressed in relational algebra as $\sigma_{ID=12345}(takes)$). If there were an index on the ID attribute of *takes*, pointers to the required records could be obtained with only a few I/O operations. Since students typically only take a few tens of courses, even fetching the actual records would take only a few tens of I/O operations subsequently. In contrast, in the absence of this index, the database system would be forced to read all *takes* records and select those with matching ID values. Reading an entire relation can be very expensive if there are a large number of students.

However, indices do have a cost, since they have to be updated whenever there is an update to the underlying relation. Creating too many indices would slow down update processing, since each update would have to also update all affected indices.

Sometimes performance problems are apparent during testing, for example, if a query takes tens of seconds, it is clear that it is quite slow. However, suppose each query takes 1 second to scan a large relation without an index, versus 10 milliseconds to retrieve the same records using an index. If testers run one query at a time, queries

Telegram: @uni_k

www.konkur.in

respond quickly, even without an index. However, suppose that the queries are part of a registration system that is used by a thousand students in an hour, and the actions of each student require 10 such queries to be executed. The total execution time would then be 10,000 seconds for queries submitted in 1 hour, that is, 3600 seconds. Students are then likely to find that the registration system is extremely slow, or even totally unresponsive. In contrast, if the required indices were present, the execution time required would be 100 seconds for queries submitted in 1 hour, and the performance of the registration system would be very good.

It is therefore important when building an application to figure out which indices are important for performance and to create them before the application goes live.

If a relation is declared to have a primary key, most database systems automatically create an index on the primary key. Whenever a tuple is inserted into the relation, the index can be used to check that the primary-key constraint is not violated (i.e., there are no duplicates on the primary-key value). Without the index on the primary key, whenever a tuple is inserted, the entire relation has to be scanned to ensure that the primary-key constraint is satisfied.

Although most database systems do not automatically create them, it is often a good idea to create indices on foreign-key attributes, too. Most joins are between foreign-key and primary-key attributes, and queries containing such joins, where there is also a selection condition on the referenced table, are not uncommon. Consider a query $takes \bowtie \sigma_{name=\text{Shankar}}(student)$, where the foreign-key attribute ID of *takes* references the primary-key attribute ID of student. Since very few students are likely to be named Shankar, the index on the foreign-key attribute *takes*.ID can be used to efficiently retrieve the *takes* tuples corresponding to these students.

Many database systems provide tools that help database administrators track what queries and updates are being executed on the system and recommend the creation of indices depending on the frequencies of the queries and updates. Such tools are referred to as index tuning wizards or advisors.

Some recent cloud-based database systems also support completely automated creation of indices whenever the system finds that doing so would avoid repeated relation scans, without the intervention of a database administrator.

14.8 Write-Optimized Index Structures

One of the drawbacks of the B^+-tree index structure is that performance can be quite poor with random writes. Consider an index that is too large to fit in memory; since the bulk of the space is at the leaf level, and memory sizes are quite large these days, we assume for simplicity that higher levels of the index fit in memory.

Now suppose writes or inserts are done in an order that does not match the sort order of the index. Then, each write/insert is likely to touch a different leaf node; if the number of leaf nodes is significantly larger than the buffer size, most of these leaf accesses would require a random read operation, as well as a subsequent write operation

Telegram: @uni_k

www.konkur.in

to write the updated leaf page back to disk. On a system with a magnetic disk, with a 10-millisecond access time, the index would support not more than 100 writes/inserts per second per disk; and this is an optimistic estimate, assuming that the seek takes the bulk of the time, and the head has not moved between the read and the write of a leaf page. On a system with flash based SSDs, random I/O is much faster, but a page write still has a significant cost since it (eventually) requires a page erase, which is an expensive operation. Thus, the basic B^+-tree structure is not ideal for applications that need to support a very large number of random writes/inserts per second.

Several alternative index structures have been proposed to handle workloads with a high write/insert rate. The **log-structured merge tree** or LSM tree and its variants are write-optimized index structures that have seen very significant adoption. The buffer tree is an alternative approach, which can be used with a variety of search tree structures. We outline these structures in the rest of this section.

14.8.1 LSM Trees

An LSM tree consists of several B^+-trees, starting with an in-memory tree, called L_0, and on-disk trees $L_1, L_2, \ldots, L_k$ for some k, where k is called the level. Figure 14.26 depicts the structure of an LSM tree for $k = 3$.

An index lookup is performed by using separate lookup operations on each of the trees $L_0, \ldots, L_k$, and merging the results of the lookups. (We assume for now that there are only inserts, and no updates or deletes; index lookups in the presence of updates/deletes are more complicated and are discussed later.)

When a record is first inserted into an LSM tree, it is inserted into the in-memory B^+-tree structure L_0. A fairly large amount of memory space is allocated for this tree. The tree grows as more inserts are processed, until it fills the memory allocated to it. At this point, we need to move data from the in-memory structure to a B^+-tree on disk.

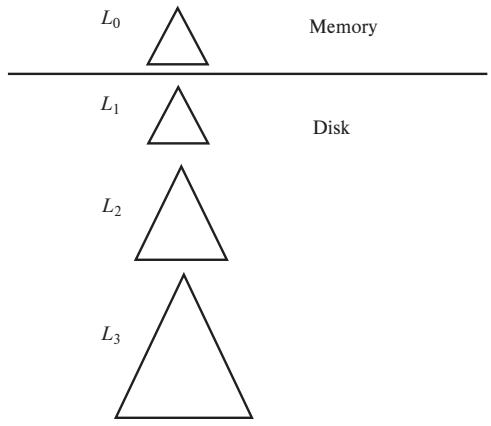


Figure 14.26 Log-structured merge tree with three levels.

Telegram: @uni_k

www.konkur.in

If tree L_1 is empty, the entire in-memory tree L_0 is written to disk to create the initial tree L_1. However, if L_1 is not empty, the leaf level of L_0 is scanned in increasing key order, and entries are merged with the leaf level entries of L_1 (also scanned in increasing key order). The merged entries are used to create a new B^+-tree, using the bottom-up build process. The new tree with the merged entries then replaces the old L_1. In either case, after entries of L_0 have been moved to L_1, all entries in L_0 as well as the old L_1, if it existed, are deleted. Inserts can then be made to the now empty L_0 in-memory.

Note that all entries in the leaf level of the old L_1 tree, including those in leaf nodes that do not have any updates, are copied to the new tree instead of performing updates on the existing L_1 tree node. This gives the following benefits.

1. The leaves of the new tree are sequentially located, avoiding random I/O during subsequent merges.
2. The leaves are full, avoiding the overhead of partially occupied leaves that can occur with page splits.

There is, however, a cost to using the LSM structure as described above: the entire contents of the tree are copied each time a set of entries from L_0 are copied into L_1. One of two techniques is used to reduce this cost:

1. Multiple levels are used, with level L_{i+1} trees having a maximum size that is k times the maximum size of level L_i trees. Thus, each record is written at most k times at a particular level. The number of levels is proportional $\log_k(I/M)$ where I is the number of entries and M is the number of entries that fit in the in-memory tree L_0.
2. Each level (other than L_0) can have up to some number b of trees, instead of just 1 tree. When an L_0 tree is written to disk, a new L_1 tree is created instead of merging it with an existing L_1 tree. When there are b such L_1 trees, they are merged into a single new L_2 tree. Similarly, when there are b trees at level L_i they are merged into a new L_{i+1} tree.

   This variant of the LSM tree is called a **stepped-merge index**. The stepped-merge index decreases the insert cost significantly compared to having only one tree per level, but it can result in an increase in query cost, since multiple trees may need to be searched. Bitmap-based structures called *Bloom filters*, described in Section 24.1, are used to reduce the number of lookups by efficiently detecting that a search key is not present in a particular tree. Bloom filters occupy very little space, but they are quite effective at reducing query cost.

Details of all these variants of LSM trees can be found in Section 24.2.

So far we have only described inserts and lookups. Deletes are handled in an interesting manner. Instead of directly finding an index entry and deleting it, deletion

Telegram: @uni_k

www.konkur.in

results in insertion of a new **deletion entry** that indicates which index entry is to be deleted. The process of inserting a deletion entry is identical to the process of inserting a normal index entry.

However, lookups have to carry out an extra step. As mentioned earlier, lookups retrieve entries from all the trees and merge them in sorted order of key value. If there is a deletion entry for some entry, both of them would have the same key value. Thus, a lookup would find both the deletion entry and the original entry for that key, which is to be deleted. If a deletion entry is found, the to-be-deleted entry is filtered out and not returned as part of the lookup result.

When trees are merged, if one of the trees contains an entry, and the other had a matching deletion entry, the entries get matched up during the merge (both would have the same key), and are both discarded.

Updates are handled in a manner similar to deletes, by inserting an update entry. Lookups need to match update entries with the original entries and return the latest value. The update is actually applied during a merge, when one tree has an entry and another has its matching update entry; the merge process would find a record and an update record with the same key, apply the update, and discard the update entry.

LSM trees were initially designed to reduce the write and seek overheads of magnetic disks. Flash based SSDs have a relatively low overhead for random I/O operations since they do not require seek, and thus the benefit of avoiding random I/O that LSM tree variants provide is not particularly important with SSDs.

However, recall that flash memory does not allow in-place update, and writing even a single byte to a page requires the whole page to be rewritten to a new physical location; the original location of the page needs to be erased eventually, which is a relatively expensive operation. The reduction in number of writes using LSM tree variants, as compared to traditional B^+-trees, can provide substantial performance benefits when LSM trees are used with SSDs.

A variant of the LSM tree similar to the stepped-merge index, with multiple trees in each layer, was used in Google's BigTable system, as well as in Apache HBase, the open source clone of BigTable. These systems are built on top of distributed file systems that allow appends to files but do not support updates to existing data. The fact that LSM trees do not perform in-place update made LSM trees a very good fit for these systems.

Subsequently, a large number of BigData storage systems such as Apache Cassandra, Apache AsterixDB, and MongoDB added support for LSM trees, with most implementing versions with multiple trees in each layer. LSM trees are also supported in MySQL (using the MyRocks storage engine) and in the embedded database systems SQLite4 and LevelDB.

14.8.2 Buffer Tree

The buffer tree is an alternative to the log-structured merge tree approach. The key idea behind the **buffer tree** is to associate a buffer with each internal node of a B^+-tree,

Telegram: @uni_k

www.konkur.in

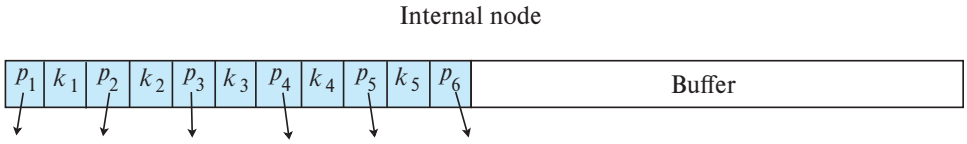


Figure 14.27 Structure of an internal node of a buffer tree.

including the root node; this is depicted pictorially in Figure 14.27. We first outline how inserts and lookups are handled, and subsequently we outline how deletes and updates are handled.

When an index record is inserted into the buffer tree, instead of traversing the tree to the leaf, the index record is inserted into the buffer of the root. If the buffer becomes full, each index record in the buffer is pushed one level down the tree to the appropriate child node. If the child node is an internal node, the index record is added to the child node's buffer; if that buffer is full, all records in that buffer are similarly pushed down. All records in a buffer are sorted on the search key before being pushed down. If the child node is a leaf node, index records are inserted into the leaf in the usual manner. If the insert results in an overfull leaf node, the node is split in the usual B^+-tree manner, with the split potentially propagating to parent nodes. Splitting of an overfull internal node is done in the usual way, with the additional step of also splitting the buffer; the buffer entries are partitioned between the two split nodes based on their key values.

Lookups are done by traversing the B^+-tree structure in the usual way, to find leaves that contain records matching the lookup key. But there is one additional step: at each internal node traversed by a lookup, the node's buffer must be examined to see if there are any records matching the lookup key. Range lookups are done as in a normal B^+-tree, but they must also examine the buffers of all internal nodes above any of the leaf nodes that are accessed.

Suppose the buffer at an internal node holds k times as many records as there are child nodes. Then, on average, k records would be pushed down at a time to each child (regardless of whether the child is an internal node or a leaf node). Sorting of records before they are pushed ensures that all these records are pushed down consecutively. The benefit of the buffer-tree approach for inserts is that the cost of accessing the child node from storage, and of writing the updated node back, is amortized (divided), on average, between k records. With sufficiently large k, the savings can be quite significant compared to inserts in a regular B^+-tree.

Deletes and updates can be processed in a manner similar to LSM trees, using deletion entries or update entries. Alternatively, deletes and updates could be processed using the normal B^+-tree algorithms, at the risk of a higher I/O cost per delete/update as compared to the cost when using deletion/update entries.

Buffer trees provide better worst-case complexity bounds on the number of I/O operations than do LSM tree variants. In terms of read cost, buffer trees are significantly faster than LSM trees. However, write operations on buffer trees involve random ,

Telegram: @uni_k

www.konkur.in

requiring more seeks, in contrast to sequential I/O operations with LSM tree variants. For magnetic disk storage, the high cost of seeks results in buffer trees performing worse than LSM trees on write-intensive workloads. LSM trees have thus found greater acceptance for write-intensive workloads with data stored on magnetic disk. However, since random I/O operations are very efficient on SSDs, and buffer trees tend to perform fewer write operations overall compared to LSM trees, buffer trees can provide better write performance on SSDs. Several index structures designed for flash storage make use of the buffer concept introduced by buffer trees.

Another benefit of buffer trees is that the key idea of associating buffers with internal nodes, to reduce the number of writes, can be used with any type of tree-structured index. For example, buffering has been used as a way of supporting bulk loading of spatial indices such as R-trees (which we study in Section 14.10.1), as well as other types of indices, for which sorting and bottom-up construction are not applicable.

Buffer trees have been implemented as part of the **Generalized Search Tree** (**GiST**) index structure in PostgreSQL. The GiST index allows user-defined code to be executed to implement search, update, and split operations on nodes and has been used to implement R-trees and other spatial index structures.

14.9 Bitmap Indices

Bitmap indices are a specialized type of index designed for easy querying on multiple keys, although each bitmap index is built on a single key. We describe key features of bitmap indices in this section but provide further details in Section 24.3.

For bitmap indices to be used, records in a relation must be numbered sequentially, starting from, say, 0. Given a number n, it must be easy to retrieve the record numbered n. This is particularly easy to achieve if records are fixed in size and allocated on consecutive blocks of a file. The record number can then be translated easily into a block number and a number that identifies the record within the block.

Consider a relation with an attribute that can take on only one of a small number (e.g., 2 to 20) of values. For instance, consider a relation *instructor_info*, which has (in addition to an ID attribute) an attribute *gender*, which can take only values m (male) or f (female). Suppose the relation also has an attribute *income_level*, which stores the income level, where income has been broken up into five levels: $L1$: 0 – 9999, $L2$: 10,000 – 19,999, $L3$: 20,000 – 39,999, $L4$: 40,000 – 74,999, and $L5$: 75,000 – ∞. Here, the raw data can take on many values, but a data analyst has split the values into a small number of ranges to simplify analysis of the data. An instance of this relation is shown on the left side of Figure 14.28.

A **bitmap** is simply an array of bits. In its simplest form, a **bitmap index** on the attribute A of relation r consists of one bitmap for each value that A can take. Each bitmap has as many bits as the number of records in the relation. The ith bit of the bitmap for value v_j is set to 1 if the record numbered i has the value v_j for attribute A. All other bits of the bitmap are set to 0.

Telegram: @uni_k

www.konkur.in

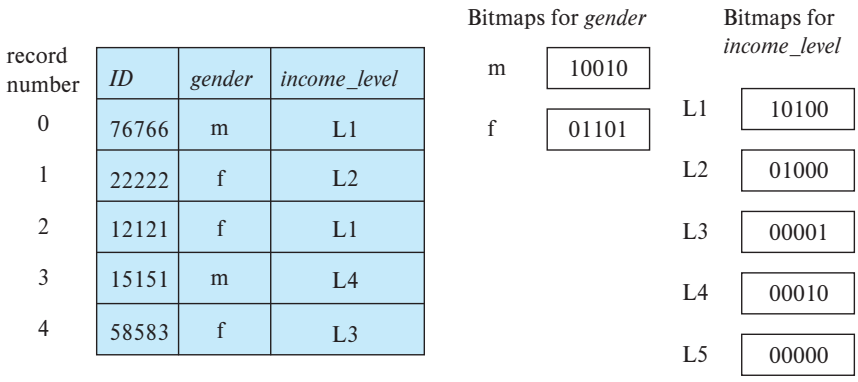


Figure 14.28 Bitmap indices on relation *instructor_info*.

In our example, there is one bitmap for the value m and one for f. The *i*th bit of the bitmap for m is set to 1 if the *gender* value of the record numbered *i* is m. All other bits of the bitmap for m are set to 0. Similarly, the bitmap for f has the value 1 for bits corresponding to records with the value f for the *gender* attribute; all other bits have the value 0. Figure 14.28 shows bitmap indices on the *gender* and *income_level* attributes of *instructor_info* relation, for the relation instance shown in the same figure.

We now consider when bitmaps are useful. The simplest way of retrieving all records with value m (or value f) would be to simply read all records of the relation and select those records with value m (or f, respectively). The bitmap index doesn't really help to speed up such a selection. While it would allow us to read only those records for a specific gender, it is likely that every disk block for the file would have to be read anyway.

In fact, bitmap indices are useful for selections mainly when there are selections on multiple keys. Suppose we create a bitmap index on attribute *income_level*, which we described earlier, in addition to the bitmap index on *gender*.

Consider now a query that selects women with income in the range 10,000 to 19,999. This query can be expressed as

```
select *
from instructor_info
where gender = 'f' and income_level = 'L2';
```

To evaluate this selection, we fetch the bitmaps for *gender* value f and the bitmap for *income_level* value *L2*, and perform an **intersection** (logical-and) of the two bitmaps. In other words, we compute a new bitmap where bit *i* has value 1 if the *i*th bit of the two bitmaps are both 1, and has a value 0 otherwise. In the example in Figure 14.28, the intersection of the bitmap for *gender* = f (01101) and the bitmap for *income_level* = *L2* (01000) gives the bitmap 01000.

Telegram: @uni_k

www.konkur.in

Since the first attribute can take two values, and the second can take five values, we would expect only about 1 in 10 records, on an average, to satisfy a combined condition on the two attributes. If there are further conditions, the fraction of records satisfying all the conditions is likely to be quite small. The system can then compute the query result by finding all bits with value 1 in the intersection bitmap and retrieving the corresponding records. If the fraction is large, scanning the entire relation would remain the cheaper alternative.

More detailed coverage of bitmap indices, including how to efficiently implement aggregate operations, how to speed up bitmap operations, and hybrid indices that combine B^+-trees with bitmaps, can be found in Section 24.3.

14.10 Indexing of Spatial and Temporal Data

Traditional index structures, such as hash indices and B^+-trees, are not suitable for indexing of spatial data, which are typically of two or more dimensions. Similarly, when tuples have temporal intervals associated with them, and queries may specify time points or time intervals, the traditional index structures may result in poor performance.

14.10.1 Indexing of Spatial Data

In this section we provide an overview of techniques for indexing spatial data. Further details can be found in Section 24.4. Spatial data refers to data referring to a point or a region in two or higher dimensional space. For example, the location of restaurants, identified by a (latitude, longitude) pair, is a form of spatial data. Similarly, the spatial extent of a farm or a lake can be identified by a polygon, with each corner identified by a (latitude, longitude) pair.

There are many forms of queries on spatial data, which need to be efficiently supported using indices. A query that asks for restaurants at a precisely specified (latitude, longitude) pair can be answered by creating a B^+-tree on the composite attribute (latitude, longitude). However, such a B^+-tree index cannot efficiently answer a query that asks for all restaurants that are within a 500-meter radius of a user's location, which is identified by a (latitude, longitude) pair. Nor can such an index efficiently answer a query that asks for all restaurants that are within a rectangular region of interest. Both of these are forms of **range queries**, which retrieve objects within a specified area. Nor can such an index efficiently answer a query that asks for the nearest restaurant to a specified location; such a query is an example of a **nearest neighbor** query.

The goal of spatial indexing is to support different forms of spatial queries, with range and nearest neighbor queries being of particular interest, since they are widely used.

To understand how to index spatial data consisting of two or more dimensions, we consider first the indexing of points in one-dimensional data. Tree structures, such as binary trees and B^+-trees, operate by successively dividing space into smaller parts. For

Telegram: @uni_k

www.konkur.in

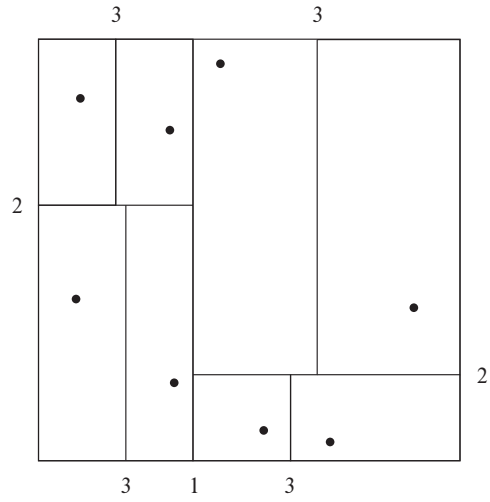


Figure 14.29 Division of space by a k-d tree.

instance, each internal node of a binary tree partitions a one-dimensional interval in two. Points that lie in the left partition go into the left subtree; points that lie in the right partition go into the right subtree. In a balanced binary tree, the partition is chosen so that approximately one-half of the points stored in the subtree fall in each partition. Similarly, each level of a B^+-tree splits a one-dimensional interval into multiple parts.

We can use that intuition to create tree structures for two-dimensional space as well as in higher-dimensional spaces. A tree structure called a **k-d tree** was one of the early structures used for indexing in multiple dimensions. Each level of a k-d tree partitions the space into two. The partitioning is done along one dimension at the node at the top level of the tree, along another dimension in nodes at the next level, and so on, cycling through the dimensions. The partitioning proceeds in such a way that, at each node, approximately one-half of the points stored in the subtree fall on one side and one-half fall on the other. Partitioning stops when a node has less than a given maximum number of points.

Figure 14.29 shows a set of points in two-dimensional space, and a k-d tree representation of the set of points, where the maximum number of points in a leaf node has been set at 1. Each line in the figure (other than the outside box) corresponds to a node in the k-d tree. The numbering of the lines in the figure indicates the level of the tree at which the corresponding node appears.

Rectangular range queries, which ask for points within a specified rectangular region, can be answered efficiently using a k-d tree as follows: Such a query essentially specifies an interval on each dimension. For example, a range query may ask for all points whose x dimension lies between 50 and 80, and y dimension lies between 40 and 70. Recall that each internal node splits space on one dimension, and as in a B^+-

Telegram: @uni_k

www.konkur.in

tree. Range search can be performed by the following recursive procedure, starting at the root:

1. Suppose the node is an internal node, and let it be split on a particular dimension, say x, at a point x_i. Entries in the left subtree have x values $< x_i$, and those in the right subtree have x values $\geq x_i$. If the query range contains x_i, search is recursively performed on both children. If the query range is to the left of x_i, search is recursively performed only on the left child, and otherwise it is performed only on the right subtree.
2. If the node is a leaf, all entries that are contained in the query range are retrieved.

Nearest neighbor search is more complicated, and we shall not describe it here, but nearest neighbor queries can also be answered quite efficiently using k-d trees.

The **k-d-B tree** extends the k-d tree to allow multiple child nodes for each internal node, just as a B-tree extends a binary tree, to reduce the height of the tree. k-d-B trees are better suited for secondary storage than k-d trees. Range search as outlined above can be easily extended to k-d-B trees, and nearest neighbor queries too can be answered quite efficiently using k-d-B trees.

There are a number of alternative index structures for spatial data. Instead of dividing the data one dimension at a time, **quadtrees** divide up a two-dimensional space into four quadrants at each node of the tree. Details may be found in Section 24.4.1.

Indexing of regions of space, such as line segments, rectangles, and other polygons, presents new problems. There are extensions of k-d trees and quadtrees for this task. A key idea is that if a line segment or polygon crosses a partitioning line, it is split along the partitioning line and represented in each of the subtrees in which its pieces occur. Multiple occurrences of a line segment or polygon can result in inefficiencies in storage, as well as inefficiencies in querying.

A storage structure called an **R-tree** is useful for indexing of objects spanning regions of space, such as line segments, rectangles, and other polygons, in addition to points. An R-tree is a balanced tree structure with the indexed objects stored in leaf nodes, much like a B^+-tree. However, instead of a range of values, a rectangular **bounding box** is associated with each tree node. The bounding box of a leaf node is the smallest rectangle parallel to the axes that contains all objects stored in the leaf node. The bounding box of internal nodes is, similarly, the smallest rectangle parallel to the axes that contains the bounding boxes of its child nodes. The bounding box of an object (such as a polygon) is defined, similarly, as the smallest rectangle parallel to the axes that contains the object.

Each internal node stores the bounding boxes of the child nodes along with the pointers to the child nodes. Each leaf node stores the indexed objects.

Figure 14.30 shows an example of a set of rectangles (drawn with a solid line) and the bounding boxes (drawn with a dashed line) of the nodes of an R-tree for the set of rectangles. Note that the bounding boxes are shown with extra space inside them, to make them stand out pictorially. In reality, the boxes would be smaller and fit tightly

Telegram: @uni_k

www.konkur.in

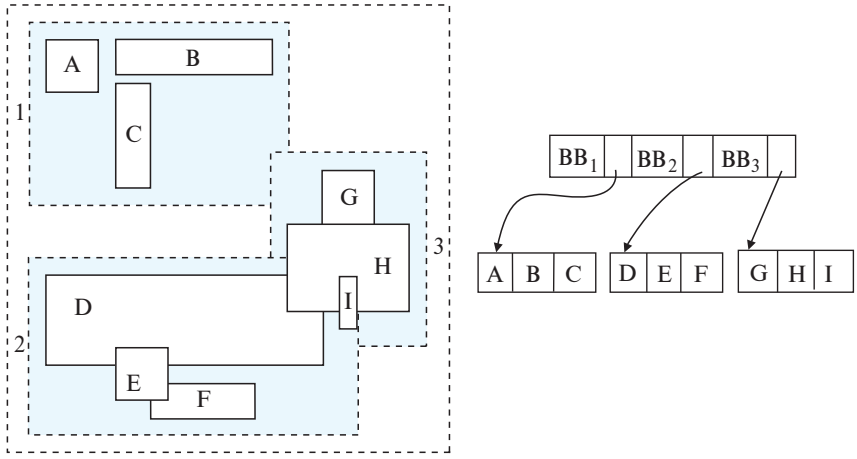


Figure 14.30 An R-tree.

on the objects that they contain; that is, each side of a bounding box B would touch at least one of the objects or bounding boxes that are contained in B.

The R-tree itself is at the right side of Figure 14.30. The figure refers to the coordinates of bounding box i as BB_i in the figure. More details about R-trees, including details of how to answer range queries using R-trees, may be found in Section 24.4.2.

Unlike some alternative structures for storing polygons and line segments, such as R*-trees and interval trees, R-trees store only one copy of each object, and we can ensure easily that each node is at least half full. However, querying may be slower than with some of the alternatives, since multiple paths have to be searched. However, because of their better storage efficiency and their similarity to B-trees, R-trees and their variants have proved popular in database systems that support spatial data.

14.10.2 Indexing Temporal Data

Temporal data refers to data that has an associated time period, as discussed in Section 7.10. The time period associated with a tuple indicates the period of time for which the tuple is valid. For example, a particular course identifier may have its title changed at some point of time. Thus, a course identifier is associated with a title for a given time interval, after which the same course identifier is associated with a different title. This can be modeled by having two or more tuples in the *course* relation with the same *course_id*, but different *title* values, each with its own valid time interval.

A **time interval** has a start time and an end time. Further a time interval indicates whether the interval starts at the start time, or just after the start time, that is, whether the interval is **closed** or **open** at the start time. Similarly, the time interval indicates whether it is closed or open at the end time. To represent the fact that a tuple is valid currently, until it is next updated, the end time is conceptually set to infinity (which can be represented by a suitably large time, such as midnight of 9999-12-31).

Telegram: @uni_k

www.konkur.in

In general, the valid period for a particular fact may not consist of just one time interval; for example, a student may be registered in a university one academic year, take a leave of absence for the next year, and register again the following year. The valid period for the student's registration at the university is clearly not a single time interval. However, any valid period can be represented by multiple intervals; thus, a tuple with any valid period can be represented by multiple tuples, each of which has a valid period that is a single time interval. We shall therefore only consider time intervals when modeling temporal data.

Suppose we wish to retrieve the value of a tuple, given a value v for an attribute a, and a point in time t_1. We can create an index on the a, and use it to retrieve all tuples with value v for attribute a. While such an index may be adequate if the number of time intervals for that search-key value is small, in general the index may retrieve a number of tuples whose time intervals do not include the time point t_1.

A better solution is to use a spatial index such as an R-tree, with the indexed tuple treated as having two dimensions, one being the indexed attribute a, and the other being the time dimension. In this case, the tuple forms a line segment, with value v for dimension a, and the valid time interval of the tuple as interval in the time dimension.

One issue that complicates the use of a spatial index such as an R-tree is that the end time interval may be infinity (perhaps represented by a very large value), whereas spatial indices typically assume that bounding boxes are finite, and may have poor performance if bounding boxes are very large. This problem can be dealt with as follows:

- All current tuples (i.e., those with end time as infinity, which is perhaps represented by a large time value) are stored in a separate index from those tuples that have a non-infinite end time. The index on current tuples can be a B^+-tree index on (a, *start_time*), where a is the indexed attribute and *start_time* is the start time, while the index for non-current tuples would be a spatial index such as an R-tree.
- Lookups for a key value v at a point in time t_i would need to search on both indices; the search on the current-tuple index would be for tuples with $a = v$, and *start_ts* $\leq t_i$, which can be done by a simple range query. Queries with a time range can be handled similarly.

Instead of using spatial indices that are designed for multidimensional data, one can use specialized indices, such as the interval B^+-tree, that are designed to index intervals in a single dimension, and provide better complexity guarantees than R-tree indices. However, most database implementations find it simpler to use R-tree indices instead of implementing yet another type of index for time intervals.

Recall that with temporal data, more than one tuple may have the same value for a primary key, as long as the tuples with the same primary-key value have non-overlapping time intervals. Temporal indices on the primary key attribute can be used to efficiently determine if the temporal primary key constraint is violated when a new tuple is inserted or the valid time interval of an existing tuple is updated.

Telegram: @uni_k

www.konkur.in

14.11 Summary

- Many queries reference only a small proportion of the records in a file. To reduce the overhead in searching for these records, we can construct *indices* for the files that store the database.
- There are two types of indices that we can use: dense indices and sparse indices. Dense indices contain entries for every search-key value, whereas sparse indices contain entries only for some search-key values.
- If the sort order of a search key matches the sort order of a relation, an index on the search key is called a *clustering index*. The other indices are called *nonclustering* or *secondary indices*. Secondary indices improve the performance of queries that use search keys other than the search key of the clustering index. However, they impose an overhead on modification of the database.
- Index-sequential files are one of the oldest index schemes used in database systems. To permit fast retrieval of records in search-key order, records are stored sequentially, and out-of-order records are chained together. To allow fast random access, we use an index structure.
- The primary disadvantage of the index-sequential file organization is that performance degrades as the file grows. To overcome this deficiency, we can use a B^+*-tree index*.
- A B^+-tree index takes the form of a *balanced* tree, in which every path from the root of the tree to a leaf of the tree is of the same length. The height of a B^+-tree is proportional to the logarithm to the base N of the number of records in the relation, where each nonleaf node stores N pointers; the value of N is often around 50 or 100. B^+-trees are much shorter than other balanced binary-tree structures such as AVL trees, and therefore require fewer disk accesses to locate records.
- Lookup on B^+-trees is straightforward and efficient. Insertion and deletion, however, are somewhat more complicated, but still efficient. The number of operations required for lookup, insertion, and deletion on B^+-trees is proportional to the logarithm to the base N of the number of records in the relation, where each nonleaf node stores N pointers.
- We can use B^+-trees for indexing a file containing records, as well as to organize records into a file.
- B-tree indices are similar to B^+-tree indices. The primary advantage of a B-tree is that the B-tree eliminates the redundant storage of search-key values. The major disadvantages are overall complexity and reduced fanout for a given node size. System designers almost universally prefer B^+-tree indices over B-tree indices in practice.

Telegram: @uni_k

www.konkur.in

- Hashing is a widely used technique for building indices in main memory as well as in disk-based systems.
- Ordered indices such as B^+-trees can be used for selections based on equality conditions involving single attributes. When multiple attributes are involved in a selection condition, we can intersect record identifiers retrieved from multiple indices.
- The basic B^+-tree structure is not ideal for applications that need to support a very large number of random writes/inserts per second. Several alternative index structures have been proposed to handle workloads with a high write/insert rate, including the log-structured merge tree and the buffer tree.
- Bitmap indices provide a very compact representation for indexing attributes with very few distinct values. Intersection operations are extremely fast on bitmaps, making them ideal for supporting queries on multiple attributes.
- R-trees are a multidimensional extension of B-trees; with variants such as R^+-trees and R^*-trees, they have proved popular in spatial databases. Index structures that partition space in a regular fashion, such as quadtrees, help in processing spatial join queries.
- There are a number of techniques for indexing temporal data, including the use of spatial index and the interval B^+-tree specialized index.

Review Terms

- Index type
  - Ordered indices
  - Hash indices
- Evaluation factors
  - Access types
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead
- Search key
- Ordered indices
  - Ordered index
  - Clustering index
  - Primary indices;
  - Nonclustering indices
  - Secondary indices
  - Index-sequential files
- Index entry
- Index record
- Dense index
- Sparse index
- Multilevel indices
- Nonunique search key
- Composite search key
- B^+-tree index files
  - Balanced tree
  - Leaf nodes

Telegram: @uni_k

www.konkur.in

- Nonleaf nodes
- Internal nodes
- Range queries
- Node split
- Node coalesce
- Redistribute of pointers
- Uniquifier

- B^+-tree extensions
  - Prefix compression
  - Bulk loading
  - Bottom-up B^+-tree construction
- B-tree indices
- Hash file organization
  - Hash function
  - Bucket
  - Overflow chaining
  - Closed addressing
  - Closed hashing
  - Bucket overflow
  - Skew
  - Static hashing
  - Dynamic hashing
- Multiple-key access
- Covering indices
- Write-optimized index structure
  - Log-structured merge (LSM) tree
  - Stepped-merge index
  - Buffer tree
- Bitmap index
- Bitmap intersection
- Indexing of spatial data
  - Range queries
  - Nearest neighbor queries
  - k-d tree
  - k-d-B tree
  - Quadtrees
  - R-tree
  - Bounding box
- Temporal indices
- Time interval
- Closed interval
- Open interval

Practice Exercises

14.1 Indices speed query processing, but it is usually a bad idea to create indices on every attribute, and every combination of attributes, that are potential search keys. Explain why.

14.2 Is it possible in general to have two clustering indices on the same relation for different search keys? Explain your answer.

14.3 Construct a B^+-tree for the following set of key values:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Telegram: @uni_k

www.konkur.in

Assume that the tree is initially empty and values are added in ascending order. Construct B^+-trees for the cases where the number of pointers that will fit in one node is as follows:

a. Four

b. Six

c. Eight

14.4 For each B^+-tree of Exercise 14.3, show the form of the tree after each of the following series of operations:

a. Insert 9.

b. Insert 10.

c. Insert 8.

d. Delete 23.

e. Delete 19.

14.5 Consider the modified redistribution scheme for B^+-trees described on page 651. What is the expected height of the tree as a function of n?

14.6 Give pseudocode for a B^+-tree function findRangeIterator(), which is like the function findRange(), except that it returns an iterator object, as described in Section 14.3.2. Also give pseudocode for the iterator class, including the variables in the iterator object, and the next() method.

14.7 What would the occupancy of each leaf node of a B^+-tree be if index entries were inserted in sorted order? Explain why.

14.8 Suppose you have a relation r with n_r tuples on which a secondary B^+-tree is to be constructed.

a. Give a formula for the cost of building the B^+-tree index by inserting one record at a time. Assume each block will hold an average of f entries and that all levels of the tree above the leaf are in memory.

b. Assuming a random disk access takes 10 milliseconds, what is the cost of index construction on a relation with 10 million records?

c. Write pseudocode for bottom-up construction of a B^+-tree, which was outlined in Section 14.4.4. You can assume that a function to efficiently sort a large file is available.

14.9 The leaf nodes of a B^+-tree file organization may lose sequentiality after a sequence of inserts.

a. Explain why sequentiality may be lost.

Telegram: @uni_k

www.konkur.in

b. To minimize the number of seeks in a sequential scan, many databases allocate leaf pages in extents of n blocks, for some reasonably large n. When the first leaf of a B^+-tree is allocated, only one block of an n-block unit is used, and the remaining pages are free. If a page splits, and its n-block unit has a free page, that space is used for the new page. If the n-block unit is full, another n-block unit is allocated, and the first $n/2$ leaf pages are placed in one n-block unit and the remaining one in the second n-block unit. For simplicity, assume that there are no delete operations.

i. What is the worst-case occupancy of allocated space, assuming no delete operations, after the first n-block unit is full?

ii. Is it possible that leaf nodes allocated to an n-node block unit are not consecutive, that is, is it possible that two leaf nodes are allocated to one n-node block, but another leaf node in between the two is allocated to a different n-node block?

iii. Under the reasonable assumption that buffer space is sufficient to store an n-page block, how many seeks would be required for a leaf-level scan of the B^+-tree, in the worst case? Compare this number with the worst case if leaf pages are allocated a block at a time.

iv. The technique of redistributing values to siblings to improve space utilization is likely to be more efficient when used with the preceding allocation scheme for leaf blocks. Explain why.

14.10 Suppose you are given a database schema and some queries that are executed frequently. How would you use the above information to decide what indices to create?

14.11 In write-optimized trees such as the LSM tree or the stepped-merge index, entries in one level are merged into the next level only when the level is full. Suggest how this policy can be changed to improve read performance during periods when there are many reads but no updates.

14.12 What trade offs do buffer trees pose as compared to LSM trees?

14.13 Consider the *instructor* relation shown in Figure 14.1.

a. Construct a bitmap index on the attribute *salary*, dividing *salary* values into four ranges: below 50,000, 50,000 to below 60,000, 60,000 to below 70,000, and 70,000 and above.

b. Consider a query that requests all instructors in the Finance department with salary of 80,000 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

14.14 Suppose you have a relation containing the x, y coordinates and names of restaurants. Suppose also that the only queries that will be asked are of the

Telegram: @uni_k

www.konkur.in

following form: The query specifies a point and asks if there is a restaurant exactly at that point. Which type of index would be preferable, R-tree or B-tree? Why?

14.15 Suppose you have a spatial database that supports region queries with circular regions, but not nearest-neighbor queries. Describe an algorithm to find the nearest neighbor by making use of multiple region queries.

Exercises

14.16 When is it preferable to use a dense index rather than a sparse index? Explain your answer.

14.17 What is the difference between a clustering index and a secondary index?

14.18 For each B^+-tree of Exercise 14.3, show the steps involved in the following queries:

a. Find records with a search-key value of 11.

b. Find records with a search-key value between 7 and 17, inclusive.

14.19 The solution presented in Section 14.3.5 to deal with nonunique search keys added an extra attribute to the search key. What effect could this change have on the height of the B^+-tree?

14.20 Suppose there is a relation $r(A, B, C)$, with a B^+-tree index with search key (A, B).

a. What is the worst-case cost of finding records satisfying $10 < A < 50$ using this index, in terms of the number of records retrieved n_1 and the height h of the tree?

b. What is the worst-case cost of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$ using this index, in terms of the number of records n_2 that satisfy this selection, as well as n_1 and h defined above?

c. Under what conditions on n_1 and n_2 would the index be an efficient way of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$?

14.21 Suppose you have to create a B^+-tree index on a large number of names, where the maximum size of a name may be quite large (say 40 characters) and the average name is itself large (say 10 characters). Explain how prefix compression can be used to maximize the average fanout of nonleaf nodes.

14.22 Suppose a relation is stored in a B^+-tree file organization. Suppose secondary indices store record identifiers that are pointers to records on disk.

Telegram: @uni_k

www.konkur.in

a. What would be the effect on the secondary indices if a node split happened in the file organization?

b. What would be the cost of updating all affected records in a secondary index?

c. How does using the search key of the file organization as a logical record identifier solve this problem?

d. What is the extra cost due to the use of such logical record identifiers?

14.23 What trade-offs do write-optimized indices pose as compared to B^+-tree indices?

14.24 An *existence bitmap* has a bit for each record position, with the bit set to 1 if the record exists, and 0 if there is no record at that position (for example, if the record were deleted). Show how to compute the existence bitmap from other bitmaps. Make sure that your technique works even in the presence of null values by using a bitmap for the value *null*.

14.25 Spatial indices that can index spatial intervals can conceptually be used to index temporal data by treating valid time as a time interval. What is the problem with doing so, and how is the problem solved?

14.26 Some attributes of relations may contain sensitive data, and may be required to be stored in an encrypted fashion. How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?

Further Reading

B-tree indices were first introduced in [Bayer and McCreight (1972)] and [Bayer (1972)]. B^+-trees are discussed in [Comer (1979)],[Bayer and Unterauer (1977)], and [Knuth (1973)]. [Gray and Reuter (1993)] provide a good description of issues in the implementation of B^+-trees.

The log-structured merge (LSM) tree is presented in [O'Neil et al. (1996)], while the stepped merge tree is presented in [Jagadish et al. (1997)]. The buffer tree is presented in [Arge (2003)]. [Vitter (2001)] provides an extensive survey of external-memory data structures and algorithms.

Bitmap indices are described in [O'Neil and Quass (1997)]. They were first introduced in the IBM Model 204 file manager on the AS 400 platform. They provide very large speedups on certain types of queries and are today implemented on most database systems.

[Samet (2006)] and [Shekhar and Chawla (2003)] provide textbook coverage of spatial data structures and spatial databases. [Bentley (1975)] describes the k-d tree,

Telegram: @uni_k

www.konkur.in

and [Robinson (1981)] describes the k-d-B tree. The R-tree was originally presented in [Guttman (1984)].

Bibliography

[Arge (2003)] L. Arge, "The Buffer Tree: A Technique for Designing Batched External Data Structures", *Algorithmica*, Volume 37, Number 1 (2003), pages 1–24.

[Bayer (1972)] R. Bayer, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica*, Volume 1, Number 4 (1972), pages 290–306.

[Bayer and McCreight (1972)] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices", *Acta Informatica*, Volume 1, Number 3 (1972), pages 173–189.

[Bayer and Unterauer (1977)] R. Bayer and K. Unterauer, "Prefix B-trees", *ACM Transactions on Database Systems*, Volume 2, Number 1 (1977), pages 11–26.

[Bentley (1975)] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, Volume 18, Number 9 (1975), pages 509–517.

[Comer (1979)] D. Comer, "The Ubiquitous B-tree", *ACM Computing Surveys*, Volume 11, Number 2 (1979), pages 121–137.

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Guttman (1984)] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), pages 47–57.

[Jagadish et al. (1997)] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental Organization for Data Recording and Warehousing", In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97 (1997), pages 16–25.

[Knuth (1973)] D. E. Knuth, *The Art of Computer Programming, Volume 3*, Addison Wesley, Sorting and Searching (1973).

[O'Neil and Quass (1997)] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1997), pages 38–49.

[O'Neil et al. (1996)] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-structured Merge-tree (LSM-tree)", *Acta Inf.*, Volume 33, Number 4 (1996), pages 351–385.

[Robinson (1981)] J. Robinson, "The k-d-B Tree: A Search Structure for Large Multidimensional Indexes", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), pages 10–18.

[Samet (2006)] H. Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann (2006).

Telegram: @uni_k

www.konkur.in

[Shekhar and Chawla (2003)] S. Shekhar and S. Chawla, *Spatial Databases: A TOUR*, Pearson (2003).

[Vitter (2001)] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data", *ACM Computing Surveys*, Volume 33, (2001), pages 209–271.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



PART 6

QUERY PROCESSING AND OPTIMIZATION

User queries have to be executed on the database contents, which reside on storage devices. It is usually convenient to break up queries into smaller operations, roughly corresponding to the relational-algebra operations. Chapter 15 describes how queries are processed, presenting algorithms for implementing individual operations and then outlining how the operations are executed in synchrony to process a query. The algorithms covered include those that can work on data much larger than main-memory, as well as those that are optimized for in-memory data.

There are many alternative ways of processing a query, and these can have widely varying costs. Query optimization refers to the process of finding the lowest-cost method of evaluating a given query. Chapter 16 describes the process of query optimization, covering techniques for estimating query plan cost, and techniques for generating alternative query plans and picking the lowest cost plans. The chapter also describes other optimization techniques, such as materialized views, for speeding up query processing.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 15



Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

15.1 Overview

The steps involved in processing a query appear in Figure 15.1. The basic steps are:

1. Parsing and translation.
2. Optimization.
3. Evaluation.

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but it is ill suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.[1] Most compiler texts cover parsing in detail.

[1] For materialized views, the expression defining the view has already been evaluated and stored. Therefore, the stored relation can be used, instead of uses of the view being replaced by the expression defining the view. Recursive views are handled differently, via a fixed-point procedure, as discussed in

Telegram: @uni_k

www.konkur.in

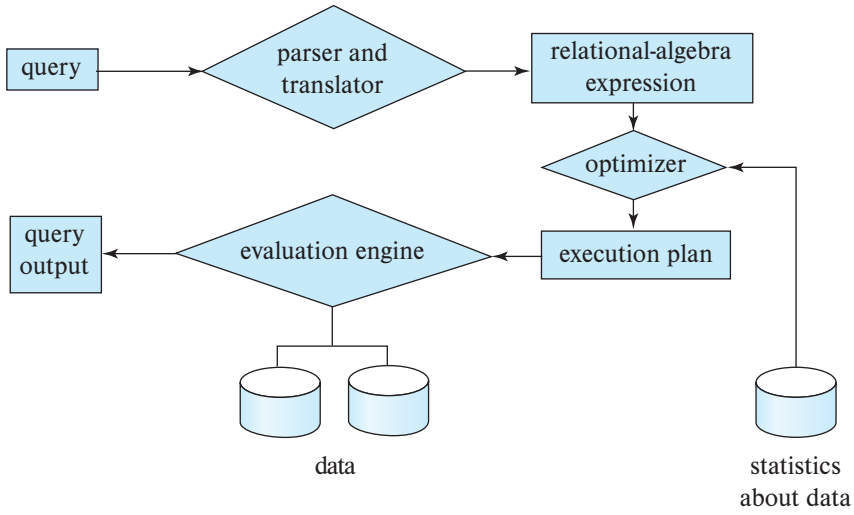


Figure 15.1 Steps in query processing.

Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relational-algebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions. As an illustration, consider the query:

select *salary*
from *instructor*
where *salary* < 75000;

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{salary<75000}\ (\Pi_{salary}\ (instructor))$
- $\Pi_{salary}\ (\sigma_{salary<75000}\ (instructor))$

Further, we can execute each relational-algebra operation by one of several different algorithms. For example, to implement the preceding selection, we can search every tuple in *instructor* to find tuples with salary less than 75000. If a B^+-tree index is available on the attribute *salary*, we can use the index instead to locate the tuples.

To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotations may state the algorithm to be used for a specific operation or the particular index or indices to use. A relational-algebra operation annotated

Telegram: @uni_k

www.konkur.in

with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**. Figure 15.2 illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it is the responsibility of the system to construct a query-evaluation plan that minimizes the cost of query evaluation; this task is called *query optimization*. Chapter 16 describes query optimization in detail.

Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.

The sequence of steps already described for processing a query is representative; not all databases exactly follow those steps. For instance, instead of using the relational-algebra representation, several databases use an annotated parse-tree representation based on the structure of the given SQL query. However, the concepts that we describe here form the basis of query processing in databases.

In order to optimize a query, a query optimizer must know the cost of each operation. Although the exact cost is hard to compute, since it depends on many parameters such as actual memory available to the operation, it is possible to get a rough estimate of execution cost for each operation.

In this chapter, we study how to evaluate individual operations in a query plan and how to estimate their cost; we return to query optimization in Chapter 16. Section 15.2 outlines how we measure the cost of a query. Section 15.3 through Section 15.6 cover the evaluation of individual relational-algebra operations. Several operations may be grouped together into a **pipeline**, in which each of the operations starts working on its input tuples even as they are being generated by another operation. In Section 15.7, we examine how to coordinate the execution of multiple operations in a query evaluation

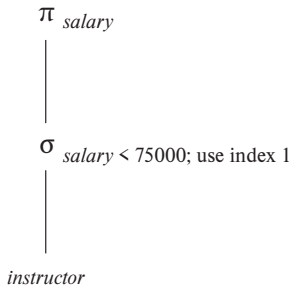


Figure 15.2 A query-evaluation plan.

Telegram: @uni_k

www.konkur.in

plan, in particular, how to use pipelined operations to avoid writing intermediate results to disk.

15.2 Measures of Query Cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations and combine them to get the cost of a query evaluation plan. Thus, as we study evaluation algorithms for each operation later in this chapter, we also outline how to estimate the cost of the operation.

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in parallel and distributed database systems, the cost of communication. (We discuss parallel and distributed database systems in Chapter 21 through Chapter 23.)

For large databases resident on magnetic disk, the I/O cost to access data from disk usually dominates the other costs; thus, early cost models focused on the I/O cost when estimating the cost of query operations. However, with flash storage becoming larger and less expensive, most organizational data today can be stored on solid-state drives (SSDs) in a cost effective manner. In addition, main memory sizes have increased significantly, and the cost of main memory has decreased enough in recent years that for many organizations, organizational data can be stored cost-effectively in main memory for querying, although it must of course be stored on flash or magnetic storage to ensure persistence.

With data resident in-memory or on SSDs, I/O cost does not dominate the overall cost, and we must include CPU costs when computing the cost of query evaluation. We do not include CPU costs in our model to simplify our presentation, but note that they can be approximated by simple estimators. For example, the cost model used by PostgreSQL (as of 2018) includes (i) a CPU cost per tuple, (ii) a CPU cost for processing each index entry (in addition to the I/O cost), and (iii) a CPU cost per operator or function (such as arithmetic operators, comparison operators, and related functions). The database has default values for each of these costs, which are multiplied by the number of tuples processed, the number of index entries processed, and the number of operators and functions executed, respectively. The defaults can be changed as a configuration parameter.

We use the *number of blocks transferred* from storage and the *number of random I/O accesses*, each of which will require a disk seek on magnetic storage, as two important factors in estimating the cost of a query-evaluation plan. If the disk subsystem takes an average of t_T seconds to transfer a block of data and has an average block-access time (disk seek time plus rotational latency) of t_S seconds, then an operation that transfers b blocks and performs S random I/O accesses would take $b * t_T + S * t_S$ seconds.

The values of t_T and t_S must be calibrated for the disk system used. We summarize performance data here; see Chapter 12 for full details on storage systems. Typical values for high-end magnetic disks in the year 2018 would be $t_S = 4$ milliseconds and $t \quad = 0.1$

Telegram: @uni_k

www.konkur.in

milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.[2]

Although SSDs do not perform a physical seek operation, they have an overhead for initiating an I/O operation; we refer to the latency from the time an I/O request is made to the time when the first byte of data is returned as t_S. For mid-range SSDs in 2018 using the SATA interface, t_S is around 90 microseconds, while the transfer time t_T is about 10 microseconds for a 4-kilobyte block. Thus, SSDs can support about 10,000 random 4-kilobyte reads per second, and they support 400 megabytes/second throughput on sequential reads using the standard SATA interface. SSDs using the PCIe 3.0x4 interface have smaller t_S, of 20 to 60 microseconds, and much higher transfer rates of around 2 gigabytes/second, corresponding to t_T of 2 microseconds, allowing around 50,000 to 15,000 random 4-kilobyte block reads per second, depending on the model.[3]

For data that are already present in main memory, reads happen at the unit of cache lines, instead of disk blocks. But assuming entire blocks of data are read, the time to transfer t_T for a 4-kilobyte block is less than 1 microsecond for data in memory. The latency to fetch data from memory, t_S, is less than 100 nanoseconds.

Given the wide diversity of speeds of different storage devices, database systems must ideally perform test seeks and block transfers to estimate t_S and t_T for specific systems/storage devices, as part of the software installation process. Databases that do not automatically infer these numbers often allow users to specify the numbers as part of configuration files.

We can refine our cost estimates further by distinguishing block reads from block writes. Block writes are typically about twice as expensive as reads on magnetic disks, since disk systems read sectors back after they are written to verify that the write was successful. On PCIe flash, write throughput may be about 50 percent less than read throughput, but the difference is almost completely masked by the limited speed of SATA interfaces, leading to write throughput matching read throughput. However, the throughput numbers do not reflect the cost of erases that are required if blocks are overwritten. For simplicity, we ignore this detail.

The cost estimates we give do not include the cost of writing the final result of an operation back to disk. These are taken into account separately where required.

[2]Storage device specifications often mention the transfer rate, and the number of random I/O operations that can be carried out in 1 second. The values t_T can be computed as block size divided by transfer rate, while t_S can be computed as $(1/N) - t_T$, where N is the number of random I/O operations per second that the device supports, since a random I/O operation performs a random I/O access, followed by data transfer of 1 block.

[3]The I/O operations per second number used here are for the case of sequential I/O requests, usually denoted as QD-1 in the SSD specifications. SSDs can support multiple random requests in parallel, with 32 to 64 parallel requests being commonly supported; an SSD with SATA interface supports nearly 100,000 random 4-kilobyte block reads in a second if multiple requests are sent in parallel, while PCIe disks can support over 350,000 random 4-kilobyte block reads per second; these numbers are referred to as the QD-32 or QD-64 numbers depending on how many requests are sent in parallel. We do not explore parallel requests in our cost model, since we only consider sequential query processing algorithms in this chapter. Shared-memory parallel query processing techniques, discussed in Section 22.6, can be used to exploit the parallel request capabilities of SSDs.

Telegram: @uni_k

www.konkur.in

The costs of all the algorithms that we consider depend on the size of the buffer in main memory. In the best case, if data fits in the buffer, the data can be read into the buffers, and the disk does not need to be accessed again. In the worst case, we may assume that the buffer can hold only a few blocks of data—approximately one block per relation. However, with large main memories available today, such worst-case assumptions are overly pessimistic. In fact, a good deal of main memory is typically available for processing a query, and our cost estimates use the amount of memory available to an operator, M, as a parameter. In PostgreSQL the total memory available to a query, called the effective cache size, is assumed by default to be 4 gigabytes, for the purpose of cost estimation; if a query has several operators that run concurrently, the available memory has to be divided amongst the operators.

In addition, although we assume that data must be read from disk initially, it is possible that a block that is accessed is already present in the in-memory buffer. Again, for simplicity, we ignore this effect; as a result, the actual disk-access cost during the execution of a plan may be less than the estimated cost. To account (at least partially) for buffer residence, PostgreSQL uses the following "hack": the cost of a random page read is assumed to be $1/10^{th}$ of the actual random page read cost, to model the situation that 90% of reads are found to be resident in cache. Further, to model the situation that internal nodes of B^+-tree indices are traversed often, most database systems assume that all internal nodes are present in the in-memory buffer, and assume that a traversal of an index only incurs a single random I/O cost for the leaf node.

The **response time** for a query-evaluation plan (that is, the wall-clock time required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan. Unfortunately, the response time of a plan is very hard to estimate without actually executing the plan, for the following two reasons:

1. The response time depends on the contents of the buffer when the query begins execution; this information is not available when the query is optimized and is hard to account for even if it were available.
2. In a system with multiple disks, the response time depends on how accesses are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.

Interestingly, a plan may get a better response time at the cost of extra resource consumption. For example, if a system has multiple disks, a plan A that requires extra disk reads, but performs the reads in parallel across multiple disks may, finish faster than another plan B that has fewer disk reads, but performs reads from only one disk at a time. However, if many instances of a query using plan A run concurrently, the overall response time may actually be more than if the same instances are executed using plan B, since plan A generates more load on the disks.

As a result, instead of trying to minimize the response time, optimizers generally try to minimize the total **resource consumption** of a query plan. Our model of estimating

Telegram: @uni_k

www.konkur.in

the total disk access time (including seek and data transfer) is an example of such a resource consumption–based model of query cost.

15.3 Selection Operation

In query processing, the **file scan** is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

15.3.1 Selections Using File Scans and Indices

Consider a selection operation on a relation whose tuples are stored together in one file. The most straightforward way of performing a selection is as follows:

- **A1** (**linear search**). In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file. In case blocks of the file are not stored contiguously, extra seeks may be required, but we ignore this effect for simplicity.

  Although it may be slower than other algorithms for implementing selection, the linear-search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation. The other algorithms that we shall study are not applicable in all cases, but when applicable they are generally faster than linear search.

Cost estimates for linear scan, as well as for other selection algorithms, are shown in Figure 15.3. In the figure, we use h_i to represent the height of the B^+-tree, and assume a random I/O operation is required for each node in the path from the root to a leaf. Most real-life optimizers assume that the internal nodes of the tree are present in the in-memory buffer since they are frequently accessed, and usually less than 1 percent of the nodes of a B^+-tree are nonleaf nodes. The cost formulae can be correspondingly simplified, charging only one random I/O cost for a traversal from the root to a leaf, by setting $h_i = 1$.

Index structures are referred to as **access paths**, since they provide a path through which data can be located and accessed. In Chapter 14, we pointed out that it is efficient to read the records of a file in an order corresponding closely to physical order. Recall that a *clustering index* (also referred to as a *primary index*) is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. An index that is not a clustering index is called a *secondary index* or a *nonclustering index*.

Telegram: @uni_k

www.konkur.in

Search algorithms that use an index are referred to as **index scans**. We use the selection predicate to guide us in the choice of the index to use in processing the query. Search algorithms that use an index are:

A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies the condition, scan can be terminated as soon as the required record is found. In the worst case, b_r block transfers are still required.
A2	Clustering B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Clustering B^+-tree Index, Equality on Non-key	$h_i * (t_T + t_S) + t_S + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a clustering index) and don't require additional seeks.
A4	Secondary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to clustering index.
A4	Secondary B^+-tree Index, Equality on Non-key	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Clustering B^+-tree Index, Comparison	$h_i * (t_T + t_S) + t_S + b * t_T$	Identical to the case of A3, equality on non-key.
A6	Secondary B^+-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on non-key.

Figure 15.3 Cost estimates for selection algorithms.

Telegram: @uni_k

www.konkur.in

- **A2 (clustering index, equality on key)**. For an equality comparison on a key attribute with a clustering index, we can use the index to retrieve a single record that satisfies the corresponding equality condition. Cost estimates are shown in Figure 15.3. To model the common situation that the internal nodes of the index are in the in-memory buffer, h_i can be set to 1.
- **A3 (clustering index, equality on non-key)**. We can retrieve multiple records by using a clustering index when the selection condition specifies an equality comparison on a non-key attribute, A. The only difference from the previous case is that multiple records may need to be fetched. However, the records must be stored consecutively in the file since the file is sorted on the search key. Cost estimates are shown in Figure 15.3.
- **A4 (secondary index, equality)**. Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may be retrieved if the indexing field is not a key.

  In the first case, only one record is retrieved. The cost in this case is the same as that for a clustering index (case A2).

  In the second case, each record may be resident on a different block, which may result in one I/O operation per retrieved record, with each I/O operation requiring a seek and a block transfer. The worst-case cost in this case is $(h_i + n) * (t_S + t_T)$, where n is the number of records fetched, if each record is in a different disk block, and the block fetches are randomly ordered. The worst-case cost could become even worse than that of linear search if a large number of records are retrieved.

  If the in-memory buffer is large, the block containing the record may already be in the buffer. It is possible to construct an estimate of the *average* or *expected* cost of the selection by taking into account the probability of the block containing the record already being in the buffer. For large buffers, that estimate will be much less than the worst-case estimate.

In certain algorithms, including A2, the use of a B^+-tree file organization can save one access since records are stored at the leaf level of the tree.

As described in Section 14.4.2, when records are stored in a B^+-tree file organization or other file organizations that may require relocation of records, secondary indices usually do not store pointers to the records.[4] Instead, secondary indices store the values of the attributes used as the search key in a B^+-tree file organization. Accessing a record through such a secondary index is then more expensive: First the secondary index is searched to find the B^+-tree file organization search-key values, then the B^+-tree file organization is looked up to find the records. The cost formulae described for secondary indices have to be modified appropriately if such indices are used.

[4]Recall that if B^+-tree file organizations are used to store relations, records may be moved between blocks when leaf nodes are split or merged, and when records are redistributed.

Telegram: @uni_k

www.konkur.in

15.3.2 Selections Involving Comparisons

Consider a selection of the form $\sigma_{A \leq v}(r)$. We can implement the selection either by using linear search or by using indices in one of the following ways:

- **A5 (clustering index, comparison)**. A clustering ordered index (for example, a clustering B^+-tree index) can be used when the selection condition is a comparison. For comparison conditions of the form $A > v$ or $A \geq v$, a clustering index on A can be used to direct the retrieval of tuples, as follows: For $A \geq v$, we look up the value v in the index to find the first tuple in the file that has a value of $A \geq v$. A file scan starting from that tuple up to the end of the file returns all tuples that satisfy the condition. For $A > v$, the file scan starts with the first tuple such that $A > v$. The cost estimate for this case is identical to that for case A3.

  For comparisons of the form $A < v$ or $A \leq v$, an index lookup is not required. For $A < v$, we use a simple file scan starting from the beginning of the file, and continuing up to (but not including) the first tuple with attribute $A = v$. The case of $A \leq v$ is similar, except that the scan continues up to (but not including) the first tuple with attribute $A > v$. In either case, the index is not useful.

- **A6 (secondary index, comparison)**. We can use a secondary ordered index to guide retrieval for comparison conditions involving $<$, $\leq$, $\geq$, or $>$. The lowest-level index blocks are scanned, either from the smallest value up to v (for $<$ and $\leq$), or from v up to the maximum value (for $>$ and $\geq$).

  The secondary index provides pointers to the records, but to get the actual records we have to fetch the records by using the pointers. This step may require an I/O operation for each record fetched, since consecutive records may be on different disk blocks; as before, each I/O operation requires a disk seek and a block transfer. If the number of retrieved records is large, using the secondary index may be even more expensive than using linear search. Therefore, the secondary index should be used only if very few records are selected.

As long as the number of matching tuples is known ahead of time, a query optimizer can choose between using a secondary index or using a linear scan based on the cost estimates. However, if the number of matching tuples is not known accurately at compilation time, either choice may lead to bad performance, depending on the actual number of matching tuples.

To deal with the above situation, PostgreSQL uses a hybrid algorithm that it calls a *bitmap index scan*,[5] when a secondary index is available, but the number of matching records is not known precisely. The bitmap index scan algorithm first creates a bitmap with as many bits as the number of blocks in the relation, with all bits initialized to 0. The algorithm then uses the secondary index to find index entries for matching tuples, but instead of fetching the tuples immediately, it does the following. As each index

[5]This algorithm should not be confused with a scan using a bitmap index.

Telegram: @uni_k

www.konkur.in

entry is found, the algorithm gets the block number from the index entry, and sets the corresponding bit in the bitmap to 1.

Once all index entries have been processed, the bitmap is scanned to find all blocks whose bit is set to 1. These are exactly the blocks containing matching records. The relation is then scanned linearly, but blocks whose bit is not set to 1 are skipped; only blocks whose bit is set to 1 are fetched, and then a scan within each block is used to retrieve all matching records in the block.

In the worst case, this algorithm is only slightly more expensive than linear scan, but in the best case it is much cheaper than linear scan. Similarly, in the worst case it is only slightly more expensive than using a secondary index scan to directly fetch tuples, but in the best case it is much cheaper than a secondary index scan. Thus, this hybrid algorithm ensures that performance is never much worse than the best plan for that database instance.

A variant of this algorithm collects all the index entries, and sorts them (using sorting algorithms which we study later in this chapter), and then performs a relation scan that skips blocks that do not have any matching entries. Using a bitmap as above can be cheaper than sorting the index entries.

15.3.3 Implementation of Complex Selections

So far, we have considered only simple selection conditions of the form *A op B*, where *op* is an equality or comparison operation. We now consider more complex selection predicates.

- **Conjunction:** A **conjunctive selection** is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disjunction:** A **disjunctive selection** is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

  A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

- **Negation:** The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples in r that are not in $\sigma_\theta(r)$.

We can implement a selection operation involving either a conjunction or a disjunction of simple conditions by using one of the following algorithms:

- **A7 (conjunctive selection using one index).** We first determine whether an access path is available for an attribute in one of the simple conditions. If one is, one of the

Telegram: @uni_k

www.konkur.in

selection algorithms A2 through A6 can retrieve records satisfying that condition. We complete the operation by testing, in the memory buffer, whether or not each retrieved record satisfies the remaining simple conditions.

To reduce the cost, we choose a θ_i and one of algorithms A1 through A6 for which the combination results in the least cost for $\sigma_{\theta_i}(r)$. The cost of algorithm A7 is given by the cost of the chosen algorithm.

- **A8 (conjunctive selection using composite index)**. An appropriate **composite index** (that is, an index on multiple attributes) may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly. The type of index determines which of algorithms A2, A3, or A4 will be used.

- **A9 (conjunctive selection by intersection of identifiers)**. Another alternative for implementing conjunctive selection operations involves the use of record pointers or record identifiers. This algorithm requires indices with record pointers, on the fields involved in the individual conditions. The algorithm scans each index for pointers to tuples that satisfy an individual condition. The intersection of all the retrieved pointers is the set of pointers to tuples that satisfy the conjunctive condition. The algorithm then uses the pointers to retrieve the actual records. If indices are not available on all the individual conditions, then the algorithm tests the retrieved records against the remaining conditions.

  The cost of algorithm A9 is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers. This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order. Thereby, (1) all pointers to records in a block come together, hence all selected records in the block can be retrieved using a single I/O operation, and (2) blocks are read in sorted order, minimizing disk-arm movement. Section 15.4 describes sorting algorithms.

- **A10 (disjunctive selection by union of identifiers)**. If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. We then use the pointers to retrieve the actual records.

  However, if even one of the conditions does not have an access path, we have to perform a linear scan of the relation to find tuples that satisfy the condition. Therefore, if there is even one such condition in the disjunct, the most efficient access method is a linear scan, with the disjunctive condition tested on each tuple during the scan.

The implementation of selections with negation conditions is left to you as an exercise (Practice Exercise 15.6).

Telegram: @uni_k

www.konkur.in

15.4 Sorting

Sorting of data plays an important role in database systems for two reasons. First, SQL queries can specify that the output be sorted. Second, and equally important for query processing, several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted. Thus, we discuss sorting here before discussing the join operation in Section 15.5.

We can sort a relation by building an index on the sort key and then using that index to read the relation in sorted order. However, such a process orders the relation only *logically*, through an index, rather than *physically*. Hence, the reading of tuples in the sorted order may lead to a disk access (disk seek plus block transfer) for each record, which can be very expensive, since the number of records can be much larger than the number of blocks. For this reason, it may be desirable to order the records physically.

The problem of sorting has been studied extensively, both for relations that fit entirely in main memory and for relations that are bigger than memory. In the first case, standard sorting techniques such as quick-sort can be used. Here, we discuss how to handle the second case.

15.4.1 External Sort-Merge Algorithm

Sorting of relations that do not fit in memory is called **external sorting**. The most commonly used technique for external sorting is the **external sort–merge** algorithm. We describe the external sort–merge algorithm next. Let M denote the number of blocks in the main memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

1. In the first stage, a number of sorted **runs** are created; each run is sorted but contains only some of the records of the relation.

```
i = 0;
repeat
    read M blocks of the relation, or the rest of the relation,
        whichever is smaller;
    sort the in-memory part of the relation;
    write the sorted data to run file R_i;
    i = i + 1;
until the end of the relation
```

2. In the second stage, the runs are *merged*. Suppose, for now, that the total number of runs, N, is less than M, so that we can allocate one block to each run and have space left to hold one block of output. The merge stage operates as follows:

Telegram: @uni_k

www.konkur.in

read one block of each of the N files R_i into a buffer block in memory;
repeat
    choose the first tuple (in sort order) among all buffer blocks;
    write the tuple to the output, and delete it from the buffer block;
    if the buffer block of any run R_i is empty **and not** end-of-file(R_i)
        then read the next block of R_i into the buffer block;
until all input buffer blocks are empty

The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations. The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sort–merge algorithm; it merges N runs, so it is called an **N-way merge**.

In general, if the relation is much larger than memory, there may be M or more runs generated in the first stage, and it is not possible to allocate a block for each run during the merge stage. In this case, the merge operation proceeds in multiple passes. Since there is enough memory for $M-1$ input buffer blocks, each merge can take $M-1$ runs as input.

The initial *pass* functions in this way: It merges the first $M-1$ runs (as described in item 2 above) to get a single run for the next pass. Then, it merges the next $M-1$ runs similarly, and so on, until it has processed all the initial runs. At this point, the number of runs has been reduced by a factor of $M-1$. If this reduced number of runs is still greater than or equal to M, another pass is made, with the runs created by the first pass as input. Each pass reduces the number of runs by a factor of $M-1$. The passes repeat as many times as required, until the number of runs is less than M; a final pass then generates the sorted output.

Figure 15.4 illustrates the steps of the external sort–merge for an example relation. For illustration purposes, we assume that only one tuple fits in a block ($f_r = 1$), and we assume that memory holds at most three blocks. During the merge stage, two blocks are used for input and one for output.

15.4.2 Cost Analysis of External Sort-Merge

We compute the disk-access cost for the external sort–merge in this way: Let b_r denote the number of blocks containing records of relation r. The first stage reads every block of the relation and writes them out again, giving a total of $2b_r$ block transfers. The initial number of runs is $\lceil b_r/M \rceil$. During the merge pass, reading in each run one block at a time leads to a large number of seeks; to reduce the number of seeks, a larger number of blocks, denoted b_b, are read or written at a time, requiring b_b buffer blocks to be allocated to each input run and to the output run. Then, $\lfloor M/b_b \rfloor - 1$ runs can be merged in each merge pass, decreasing the number of runs by a factor of $\lfloor M/b_b \rfloor - 1$. The total number of merge passes required is $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil$. Each of these passes reads every block of the relation once and writes it out once, with two exceptions. First, the final pass can produce the sorted output without writing its result

Telegram: @uni_k

www.konkur.in

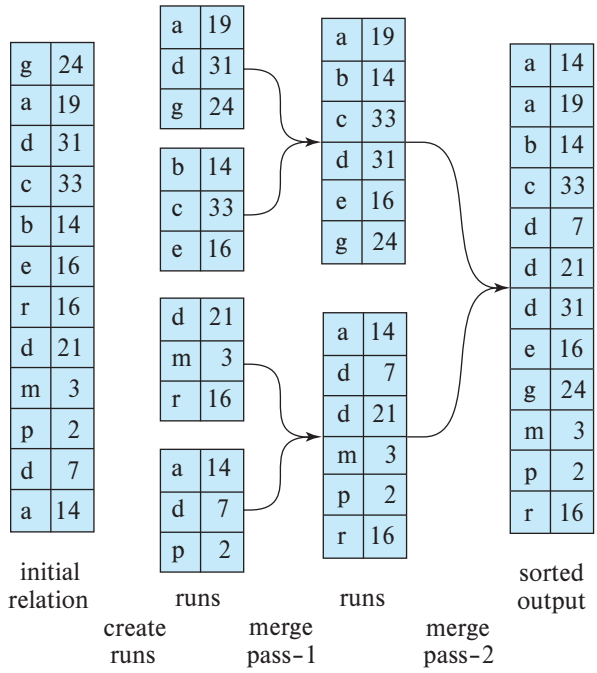


Figure 15.4 External sorting using sort-merge.

to disk. Second, there may be runs that are not read in or written out during a pass —for example, if there are $\lfloor M/b_b \rfloor$ runs to be merged in a pass, $\lfloor M/b_b \rfloor - 1$ are read in and merged, and one run is not accessed during the pass. Ignoring the (relatively small) savings due to the latter effect, the total number of block transfers for external sorting of the relation is:

$$b_r(2\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil + 1)$$

Applying this equation to the example in Figure 15.4, with b_b set to 1, we get a total of $12 * (4 + 1) = 60$ block transfers, as you can verify from the figure. Note that these above numbers do not include the cost of writing out the final result.

We also need to add the disk-seek costs. Run generation requires seeks for reading data for each of the runs as well as for writing the runs. Each merge pass requires around $\lceil b_r/b_b \rceil$ seeks for reading data.[6] Although the output is written sequentially, if it is on the same disk as the input runs, the head may have moved away between writes of consecutive blocks. Thus we would have to add a total of $2\lceil b_r/b_b \rceil$ seeks for each merge pass, except the final pass (since we assume the final result is not written back to disk).

[6]To be more precise, since we read each run separately and may get fewer than b_b blocks when reading the end of a run, we may require an extra seek for each run. We ignore this

Telegram: @uni_k

www.konkur.in

$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_r/M) \rceil - 1)$$

Applying this equation to the example in Figure 15.4, we get a total of $8 + 12 * (2 * 2 - 1) = 44$ disk seeks if we set the number of buffer blocks per run b_b to 1.

15.5 Join Operation

In this section, we study several algorithms for computing the join of relations, and we analyze their respective costs.

We use the term **equi-join** to refer to a join of the form $r \bowtie_{r.A=s.B} s$, where A and B are attributes or sets of attributes of relations r and s, respectively.

We use as a running example the expression:

$$student \bowtie takes$$

using the same relation schemas that we used in Chapter 2. We assume the following information about the two relations:

- Number of records of *student*: $n_{student} = 5000$.
- Number of blocks of *student*: $b_{student} = 100$.
- Number of records of *takes*: $n_{takes} = 10,000$.
- Number of blocks of *takes*: $b_{takes} = 400$.

15.5.1 Nested-Loop Join

Figure 15.5 shows a simple algorithm to compute the theta join, $r \bowtie_\theta s$, of two relations r and s. This algorithm is called the **nested-loop join** algorithm, since it basically consists of a pair of nested **for** loops. Relation r is called the **outer relation** and relation s the **inner relation** of the join, since the loop for r encloses the loop for s. The algorithm uses the notation $t_r \cdot t_s$, where t_r and t_s are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples t_r and t_s.

Like the linear file-scan algorithm for selection, the nested-loop join algorithm requires no indices, and it can be used regardless of what the join condition is. Extending the algorithm to compute the natural join is straightforward, since the natural join can be expressed as a theta join followed by elimination of repeated attributes by a projection. The only change required is an extra step of deleting repeated attributes from the tuple $t_r \cdot t_s$, before adding it to the result.

The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations. Consider the cost of the nested-loop join algorithm. The number of pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r, and n_s denotes the number of tuples in s. For each record in r, we have to perform

Telegram: @uni_k

www.konkur.in


```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result;
    end
end
```

Figure 15.5 Nested-loop join.

a complete scan on s. In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block transfers would be required, where b_r and b_s denote the number of blocks containing tuples of r and s, respectively. We need only one seek for each scan on the inner relation s since it is read sequentially, and a total of b_r seeks to read r, leading to a total of $n_r + b_r$ seeks. In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only $b_r + b_s$ block transfers would be required, along with two seeks.

If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation, since the inner relation would then be read only once. Therefore, if s is small enough to fit in main memory, our strategy requires only a total $b_r + b_s$ block transfers and two seeks—the same cost as that for the case where both relations fit in memory.

Now consider the natural join of *student* and *takes*. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index. We can use the nested loops to compute the join; assume that *student* is the outer relation and *takes* is the inner relation in the join. We will have to examine $5000 * 10{,}000 = 50 * 10^6$ pairs of tuples. In the worst case, the number of block transfers is $5000 * 400 + 100 = 2{,}000{,}100$, plus $5000 + 100 = 5100$ seeks. In the best-case scenario, however, we can read both relations only once and perform the computation. This computation requires at most $100 + 400 = 500$ block transfers, plus two seeks —a significant improvement over the worst-case scenario. If we had used *takes* as the relation for the outer loop and *student* for the inner loop, the worst-case cost of our final strategy would have been $10{,}000 * 100 + 400 = 1{,}000{,}400$ block transfers, plus 10,400 disk seeks. The number of block transfers is significantly less, and although the number of seeks is higher, the overall cost is reduced, assuming $t_S = 4$ milliseconds and $t_T = 0.1$ milliseconds.

15.5.2 Block Nested-Loop Join

If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather

Telegram: @uni_k

www.konkur.in

than on a per-tuple basis. Figure 15.6 shows **block nested-loop join**, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

The primary difference in cost between the block nested-loop join and the basic nested-loop join is that, in the worst case, each block in the inner relation s is read only once for each *block* in the outer relation, instead of once for each *tuple* in the outer relation. Thus, in the worst case, there will be a total of $b_r * b_s + b_r$ block transfers, where b_r and b_s denote the number of blocks containing records of r and s, respectively. Each scan of the inner relation requires one seek, and the scan of the outer relation requires one seek per block, leading to a total of $2 * b_r$ seeks. It is more efficient to use the smaller relation as the outer relation, in case neither of the relations fits in memory. In the best case, where the inner relation fits in memory, there will be $b_r + b_s$ block transfers and just two seeks (we would choose the smaller relation as the inner relation in this case).

Now return to our example of computing *student* $\bowtie$ *takes*, using the block nested-loop join algorithm. In the worst case, we have to read each block of *takes* once for each block of *student*. Thus, in the worst case, a total of $100 * 400 + 100 = 40{,}100$ block transfers plus $2 * 100 = 200$ seeks are required. This cost is a significant improvement over the $5000 * 400 + 100 = 2{,}000{,}100$ block transfers plus 5100 seeks needed in the worst case for the basic nested-loop join. The best-case cost remains the same—namely, $100 + 400 = 500$ block transfers and two seeks.

The performance of the nested-loop and block nested-loop procedures can be further improved:

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result;
            end
        end
    end
end
```

Figure 15.6 Block nested-loop join.

Telegram: @uni_k

www.konkur.in

- If the join attributes in a natural join or an equi-join form a key on the inner relation, then for each outer relation tuple the inner loop can terminate as soon as the first match is found.
- In the block nested-loop algorithm, instead of using disk blocks as the blocking unit for the outer relation, we can use the biggest size that can fit in memory, while leaving enough space for the buffers of the inner relation and the output. In other words, if memory has M blocks, we read in $M - 2$ blocks of the outer relation at a time, and when we read each block of the inner relation we join it with all the $M - 2$ blocks of the outer relation. This change reduces the number of scans of the inner relation from b_r to $\lceil b_r/(M - 2) \rceil$, where b_r is the number of blocks of the outer relation. The total cost is then $\lceil b_r/(M - 2) \rceil * b_s + b_r$ block transfers and $2\lceil b_r/(M - 2) \rceil$ seeks.
- We can scan the inner loop alternately forward and backward. This scanning method orders the requests for disk blocks so that the data remaining in the buffer from the previous scan can be reused, thus reducing the number of disk accesses needed.
- If an index is available on the inner loop's join attribute, we can replace file scans with more efficient index lookups. Section 15.5.3 describes this optimization.

15.5.3 Indexed Nested-Loop Join

In a nested-loop join (Figure 15.5), if an index is available on the inner loop's join attribute, index lookups can replace file scans. For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r.

This join method is called an **indexed nested-loop join**; it can be used with existing indices, as well as with temporary indices created for the sole purpose of evaluating the join.

Looking up tuples in s that will satisfy the join conditions with a given tuple t_r is essentially a selection on s. For example, consider *student* $\bowtie$ *takes*. Suppose that we have a *student* tuple with *ID* "00128". Then, the relevant tuples in *takes* are those that satisfy the selection "*ID* = 00128".

The cost of an indexed nested-loop join can be computed as follows: For each tuple in the outer relation r, a lookup is performed on the index for s, and the relevant tuples are retrieved. In the worst case, there is space in the buffer for only one block of r and one block of the index. Then, b_r I/O operations are needed to read relation r, where b_r denotes the number of blocks containing records of r; each I/O requires a seek and a block transfer, since the disk head may have moved in between each I/O. For each tuple in r, we perform an index lookup on s. Then, the cost of the join can be computed as $b_r(t_T + t_S) + n_r * c$, where n_r is the number of records in relation r, and c is the cost of a single selection on s using the join condition. We have seen in Section 15.3 how to estimate the cost of a single selection algorithm (possibly using indices); that estimate gives us the value of c.

Telegram: @uni_k

www.konkur.in

The cost formula indicates that, if indices are available on both relations r and s, it is generally most efficient to use the one with fewer tuples as the outer relation.

For example, consider an indexed nested-loop join of *student* $\bowtie$ *takes*, with *student* as the outer relation. Suppose also that *takes* has a clustering B^+-tree index on the join attribute *ID*, which contains 20 entries on average in each index node. Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data. Since $n_{student}$ is 5000, the total cost is $100 + 5000 * 5 = 25{,}100$ disk accesses, each of which requires a seek and a block transfer. In contrast, as we saw before, 40,100 block transfers plus 200 seeks were needed for a block nested-loop join. Although the number of block transfers has been reduced, the seek cost has actually increased, increasing the total cost since a seek is considerably more expensive than a block transfer. However, if we had a selection on the *student* relation that reduces the number of rows significantly, indexed nested-loop join could be significantly faster than block nested-loop join.

15.5.4 Merge Join

The **merge-join** algorithm (also called the **sort-merge-join** algorithm) can be used to compute natural joins and equi-joins. Let $r(R)$ and $s(S)$ be the relations whose natural join is to be computed, and let $R \cap S$ denote their common attributes. Suppose that both relations are sorted on the attributes $R \cap S$. Then, their join can be computed by a process much like the merge stage in the merge–sort algorithm.

15.5.4.1 Merge-Join Algorithm

Figure 15.7 shows the merge-join algorithm. In the algorithm, *JoinAttrs* refers to the attributes in $R \cap S$, and $t_r \bowtie t_s$, where t_r and t_s are tuples that have the same values for *JoinAttrs*, denotes the concatenation of the attributes of the tuples, followed by projecting out repeated attributes. The merge-join algorithm associates one pointer with each relation. These pointers point initially to the first tuple of the respective relations. As the algorithm proceeds, the pointers move through the relation. A group of tuples of one relation with the same value on the join attributes is read into S_s. The algorithm in Figure 15.7 *requires* that every set of tuples S_s fit in main memory; we discuss extensions of the algorithm to avoid this requirement shortly. Then, the corresponding tuples (if any) of the other relation are read in and are processed as they are read.

Figure 15.8 shows two relations that are sorted on their join attribute $a1$. It is instructive to go through the steps of the merge-join algorithm on the relations shown in the figure.

The merge-join algorithm of Figure 15.7 requires that each set S_s of all tuples with the same value for the join attributes must fit in main memory. This requirement can usually be met, even if the relation s is large. If there are some join attribute values for

Telegram: @uni_k

www.konkur.in


```
pr := address of first tuple of r;
ps := address of first tuple of s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tuple to which ps points;
        S_s := {t_s};
        set ps to point to next tuple of s;
        done := false;
        while (not done and ps ≠ null) do
            begin
                t_s' := tuple to which ps points;
                if (t_s'[JoinAttrs] = t_s[JoinAttrs])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            set ps to point to next tuple of s;
                         end
                    else done := true;
            end
        t_r := tuple to which pr points;
        while (pr ≠ null and t_r[JoinAttrs] < t_s[JoinAttrs]) do
            begin
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
        while (pr ≠ null and t_r[JoinAttrs] = t_s[JoinAttrs]) do
            begin
                for each t_s in S_s do
                    begin
                        add t_s ⋈ t_r to result;
                    end
                set pr to point to next tuple of r;
                t_r := tuple to which pr points;
            end
    end.
```

Figure 15.7 Merge join.

which S_s is larger than available memory, a block nested-loop join can be performed for such sets S_s, matching them with corresponding blocks of tuples in r with the same values for the join attributes.

Telegram: @uni_k

www.konkur.in

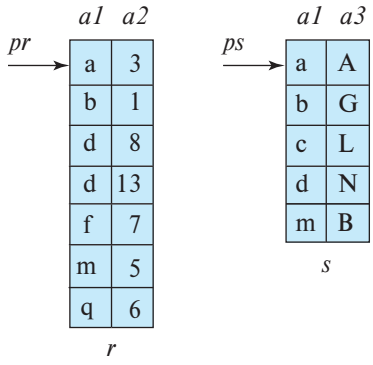


Figure 15.8 Sorted relations for merge join.

If either of the input relations r and s is not sorted on the join attributes, they can be sorted first, and then the merge-join algorithm can be used. The merge-join algorithm can also be easily extended from natural joins to the more general case of equi-joins.

15.5.4.2 Cost Analysis

Once the relations are in sorted order, tuples with the same value on the join attributes are in consecutive order. Thereby, each tuple in the sorted order needs to be read only once, and, as a result, each block is also read only once. Since it makes only a single pass through both files (assuming all sets S_s fit in memory), the merge-join method is efficient; the number of block transfers is equal to the sum of the number of blocks in both files, $b_r + b_s$.

Assuming that b_b buffer blocks are allocated to each relation, the number of disk seeks required would be $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ disk seeks. Since seeks are much more expensive than data transfer, it makes sense to allocate multiple buffer blocks to each relation, provided extra memory is available. For example, with $t_T = 0.1$ milliseconds per 4-kilobyte block, and $t_S = 4$ milliseconds, the buffer size is 400 blocks (or 1.6 megabytes), so the seek time would be 4 milliseconds for every 40 milliseconds of transfer time; in other words, seek time would be just 10 percent of the transfer time.

If either of the input relations r and s is not sorted on the join attributes, they must be sorted first; the cost of sorting must then be added to the above costs. If some sets S_s do not fit in memory, the cost would increase slightly.

Suppose the merge-join scheme is applied to our example of *student* $\bowtie$ *takes*. The join attribute here is *ID*. Suppose that the relations are already sorted on the join attribute *ID*. In this case, the merge join takes a total of $400+100 = 500$ block transfers. If we assume that in the worst case only one buffer block is allocated to each input relation (that is, $b_b = 1$), a total of $400 + 100 = 500$ seeks would also be required; in reality b_b can be set much higher since we need to buffer blocks for only two relations, and the seek cost would be significantly less.

Telegram: @uni_k

www.konkur.in

Suppose the relations are not sorted, and the memory size is the worst case, only three blocks. The cost is as follows:

1. Using the formulae that we developed in Section 15.4, we can see that sorting relation *takes* requires $\lceil \log_{3-1}(400/3) \rceil = 8$ merge passes. Sorting of relation *takes* then takes $400 * (2\lceil \log_{3-1}(400/3) \rceil + 1)$, or 6800, block transfers, with 400 more transfers to write out the result. The number of seeks required is $2 * \lceil 400/3 \rceil + 400 * (2 * 8 - 1)$ or 6268 seeks for sorting, and 400 seeks for writing the output, for a total of 6668 seeks, since only one buffer block is available for each run.
2. Similarly, sorting relation *student* takes $\lceil \log_{3-1}(100/3) \rceil = 6$ merge passes and $100 * (2\lceil \log_{3-1}(100/3) \rceil + 1)$, or 1300, block transfers, with 100 more transfers to write it out. The number of seeks required for sorting *student* is $2 * \lceil 100/3 \rceil + 100 * (2 * 6 - 1) = 1168$, and 100 seeks are required for writing the output, for a total of 1268 seeks.
3. Finally, merging the two relations takes $400 + 100 = 500$ block transfers and 500 seeks.

Thus, the total cost is 9100 block transfers plus 8932 seeks if the relations are not sorted, and the memory size is just 3 blocks.

With a memory size of 25 blocks, and the relations not sorted, the cost of sorting followed by merge join would be as follows:

1. Sorting the relation *takes* can be done with just one merge step and takes a total of just $400 * (2\lceil \log_{24}(400/25) \rceil + 1) = 1200$ block transfers. Similarly, sorting *student* takes 300 block transfers. Writing the sorted output to disk requires $400 + 100 = 500$ block transfers, and the merge step requires 500 block transfers to read the data back. Adding up these costs gives a total cost of 2500 block transfers.
2. If we assume that only one buffer block is allocated for each run, the number of seeks required in this case is $2 * \lceil 400/25 \rceil + 400 + 400 = 832$ seeks for sorting *takes* and writing the sorted output to disk, and similarly $2 * \lceil 100/25 \rceil + 100 + 100 = 208$ for *student*, plus $400 + 100$ seeks for reading the sorted data in the merge-join step. Adding up these costs gives a total cost of 1640 seeks.

   The number of seeks can be significantly reduced by setting aside more buffer blocks for each run. For example, if 5 buffer blocks are allocated for each run and for the output from merging the 4 runs of *student*, the cost is reduced to $2 * \lceil 100/25 \rceil + \lceil 100/5 \rceil + \lceil 100/5 \rceil = 48$ seeks, from 208 seeks. If the merge-join step sets aside 12 blocks each for buffering *takes* and *student*, the number of seeks for the merge-join step goes down to $\lceil 400/12 \rceil + \lceil 100/12 \rceil = 43$, from 500. The total number of seeks is then 251.

Telegram: @uni_k

www.konkur.in

Thus, the total cost is 2500 block transfers plus 251 seeks if the relations are not sorted, and the memory size is 25 blocks.

15.5.4.3 Hybrid Merge Join

It is possible to perform a variation of the merge-join operation on unsorted tuples, if secondary indices exist on both join attributes. The algorithm scans the records through the indices, resulting in their being retrieved in sorted order. This variation presents a significant drawback, however, since records may be scattered throughout the file blocks. Hence, each tuple access could involve accessing a disk block, and that is costly.

To avoid this cost, we can use a hybrid merge-join technique that combines indices with merge join. Suppose that one of the relations is sorted; the other is unsorted, but has a secondary B^+-tree index on the join attributes. The **hybrid merge-join algorithm** merges the sorted relation with the leaf entries of the secondary B^+-tree index. The result file contains tuples from the sorted relation and addresses for tuples of the unsorted relation. The result file is then sorted on the addresses of tuples of the unsorted relation, allowing efficient retrieval of the corresponding tuples, in physical storage order, to complete the join. Extensions of the technique to handle two unsorted relations are left as an exercise for you.

15.5.5 Hash Join

Like the merge-join algorithm, the hash-join algorithm can be used to implement natural joins and equi-joins. In the hash-join algorithm, a hash function h is used to partition tuples of both relations. The basic idea is to partition the tuples of each of the relations into sets that have the same hash value on the join attributes.

We assume that:

- h is a hash function mapping *JoinAttrs* values to $\{0, 1, \ldots, n_h\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
- $r_0, r_1, \ldots, r_{n_h}$ denote partitions of r tuples, each initially empty. Each tuple $t_r \in r$ is put in partition r_i, where $i = h(t_r[JoinAttrs])$.
- $s_0, s_1, \ldots, s_{n_h}$ denote partitions of s tuples, each initially empty. Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.

The hash function h should have the "goodness" properties of randomness and uniformity that we discussed in Chapter 14. Figure 15.9 depicts the partitioning of the relations.

15.5.5.1 Basics

The idea behind the hash-join algorithm is this: Suppose that an r tuple and an s tuple satisfy the join condition; then, they have the same value for the join attributes. If that value is hashed to some value i, the r tuple has to be in r_i and the s tuple in s . Therefore,

Telegram: @uni_k

www.konkur.in

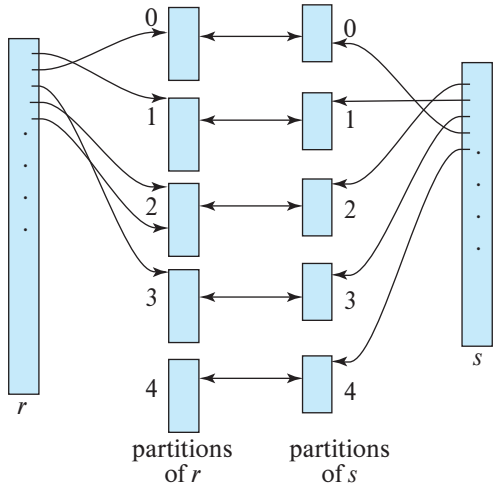


Figure 15.9 Hash partitioning of relations.

r tuples in r_i need only be compared with s tuples in s_i; they do not need to be compared with s tuples in any other partition.

For example, if d is a tuple in *student*, c a tuple in *takes*, and h a hash function on the *ID* attributes of the tuples, then d and c must be tested only if $h(c) = h(d)$. If $h(c) \neq h(d)$, then c and d must have different values for *ID*. However, if $h(c) = h(d)$, we must test c and d to see whether the values in their join attributes are the same, since it is possible that c and d have different *iid*s that have the same hash value.

Figure 15.10 shows the details of the **hash-join** algorithm to compute the natural join of relations r and s. As in the merge-join algorithm, $t_r \bowtie t_s$ denotes the concatenation of the attributes of tuples t_r and t_s, followed by projecting out repeated attributes. After the partitioning of the relations, the rest of the hash-join code performs a separate indexed nested-loop join on each of the partition pairs i, for $i = 0, \ldots, n_h$. To do so, it first **builds** a hash index on each s_i, and then **probes** (that is, looks up s_i) with tuples from r_i. The relation s is the **build input**, and r is the **probe input**.

The hash index on s_i is built in memory, so there is no need to access the disk to retrieve the tuples. The hash function used to build this hash index must be different from the hash function h used earlier, but it is still applied to only the join attributes. In the course of the indexed nested-loop join, the system uses this hash index to retrieve records that match records in the probe input.

The build and probe phases require only a single pass through both the build and probe inputs. It is straightforward to extend the hash-join algorithm to compute general equi-joins.

The value n_h must be chosen to be large enough such that, for each i, the tuples in the partition s_i of the build relation, along with the hash index on the partition, fit in memory. It is not necessary for the partitions of the probe relation to fit in memory. It is

Telegram: @uni_k

www.konkur.in


```
/* Partition s */
for each tuple t_s in s do begin
  i := h(t_s[JoinAttrs]);
  H_si := H_si ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
  i := h(t_r[JoinAttrs]);
  H_ri := H_ri ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
  read H_si and build an in-memory hash index on it;
  for each tuple t_r in H_ri do begin
    probe the hash index on H_si to locate all tuples t_s
      such that t_s[JoinAttrs] = t_r[JoinAttrs];
    for each matching tuple t_s in H_si do begin
      add t_r ⋈ t_s to the result;
    end
  end
end
```

Figure 15.10 Hash join.

best to use the smaller input relation as the build relation. If the size of the build relation is b_s blocks, then, for each of the n_h partitions to be of size less than or equal to M, n_h must be at least $\lceil b_s/M \rceil$. More precisely stated, we have to account for the extra space occupied by the hash index on the partition as well, so n_h should be correspondingly larger. For simplicity, we sometimes ignore the space requirement of the hash index in our analysis.

15.5.5.2 Recursive Partitioning

If the value of n_h is greater than or equal to the number of blocks of memory, the relations cannot be partitioned in one pass, since there will not be enough buffer blocks. Instead, partitioning has to be done in repeated passes. In one pass, the input can be split into at most as many partitions as there are blocks available for use as output buffers. Each bucket generated by one pass is separately read in and partitioned again in the next pass, to create smaller partitions. The hash function used in a pass is different from the one used in the previous pass. The system repeats this splitting of the

Telegram: @uni_k

www.konkur.in

input until each partition of the build input fits in memory. Such partitioning is called **recursive partitioning**.

A relation does not need recursive partitioning if $M > n_h + 1$, or equivalently $M > (b_s/M)+1$, which simplifies (approximately) to $M > \sqrt{b_s}$. For example, consider a memory size of 12 megabytes, divided into 4-kilobyte blocks; it would contain a total of 3-kilobyte (3072) blocks. We can use a memory of this size to partition relations of size up to 3-kilobyte $*$ 3-kilobyte blocks, which is 36 gigabytes. Similarly, a relation of size 1 gigabyte requires just over $\sqrt{256\text{K}}$ blocks, or 2 megabytes, to avoid recursive partitioning.

15.5.5.3 Handling of Overflows

Hash-table overflow occurs in partition i of the build relation s if the hash index on s_i is larger than main memory. Hash-table overflow can occur if there are many tuples in the build relation with the same values for the join attributes, or if the hash function does not have the properties of randomness and uniformity. In either case, some of the partitions will have more tuples than the average, whereas others will have fewer; partitioning is then said to be **skewed**.

We can handle a small amount of skew by increasing the number of partitions so that the expected size of each partition (including the hash index on the partition) is somewhat less than the size of memory. The number of partitions is therefore increased by a small value, called the **fudge factor**, that is usually about 20 percent of the number of hash partitions computed as described in Section 15.5.5.

Even if, by using a fudge factor, we are conservative on the sizes of the partitions, overflows can still occur. Hash-table overflows can be handled by either *overflow resolution* or *overflow avoidance*. **Overflow resolution** is performed during the build phase if a hash-index overflow is detected. Overflow resolution proceeds in this way: If s_i, for any i, is found to be too large, it is further partitioned into smaller partitions by using a different hash function. Similarly, r_i is also partitioned using the new hash function, and only tuples in the matching partitions need to be joined.

In contrast, **overflow avoidance** performs the partitioning carefully, so that overflows never occur during the build phase. In overflow avoidance, the build relation s is initially partitioned into many small partitions, and then some partitions are combined in such a way that each combined partition fits in memory. The probe relation r is partitioned in the same way as the combined partitions on s, but the sizes of r_i do not matter.

If a large number of tuples in s have the same value for the join attributes, the resolution and avoidance techniques may fail on some partitions. In that case, instead of creating an in-memory hash index and using a nested-loop join to join the partitions, we can use other join techniques, such as block nested-loop join, on those partitions.

15.5.5.4 Cost of Hash Join

We now consider the cost of a hash join. Our analysis assumes that there is no hash-table overflow. First, consider the case where recursive partitioning is not required.

Telegram: @uni_k

www.konkur.in

- The partitioning of the two relations r and s calls for a complete reading of both relations and a subsequent writing back of them. This operation requires $2(b_r + b_s)$ block transfers, where b_r and b_s denote the number of blocks containing records of relations r and s, respectively. The build and probe phases read each of the partitions once, calling for further $b_r + b_s$ block transfers. The number of blocks occupied by partitions could be slightly more than $b_r + b_s$, as a result of partially filled blocks. Accessing such partially filled blocks can add an overhead of at most $2n_h$ for each of the relations, since each of the n_h partitions could have a partially filled block that has to be written and read back. Thus, a hash join is estimated to require:

  $$3(b_r + b_s) + 4n_h$$

  block transfers. The overhead $4n_h$ is usually quite small compared to $b_r + b_s$ and can be ignored.

- Assuming b_b blocks are allocated for the input buffer and each output buffer, partitioning requires a total of $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ seeks. The build and probe phases require only one seek for each of the n_h partitions of each relation, since each partition can be read sequentially. The hash join thus requires $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$ seeks.

Now consider the case where recursive partitioning is required. Again we assume that b_b blocks are allocated for buffering each partition. Each pass then reduces the size of each of the partitions by an expected factor of $\lfloor M/b_b \rfloor - 1$; and passes are repeated until each partition is of size at most M blocks. The expected number of passes required for partitioning s is therefore $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil$.

- Since, in each pass, every block of s is read in and written out, the total number of block transfers for partitioning of s is $2b_s \lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil$. The number of passes for partitioning of r is the same as the number of passes for partitioning of s, therefore the join is estimated to require

  $$2(b_r + b_s)\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil + b_r + b_s$$

  block transfers.

- Ignoring the relatively small number of seeks during the build and probe phases, hash join with recursive partitioning requires

  $$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil$$

  disk seeks.

Consider, for example, the natural join *takes* $\bowtie$ *student*. With a memory size of 20 blocks, the *student* relation can be partitioned into five partitions, each of size 20 blocks, which size will fit into memory. Only one pass is required for the partitioning. The

Telegram: @uni_k

www.konkur.in

relation *takes* is similarly partitioned into five partitions, each of size 80. Ignoring the cost of writing partially filled blocks, the cost is $3(100 + 400) = 1500$ block transfers. There is enough memory to allocate the buffers for the input and each of the five outputs during partitioning (i.e, $b_b = 3$) leading to $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ seeks.

The hash join can be improved if the main memory size is large. When the entire build input can be kept in main memory, n_h can be set to 0; then, the hash-join algorithm executes quickly, without partitioning the relations into temporary files, regardless of the probe input's size. The cost estimate goes down to $b_r + b_s$ block transfers and two seeks.

Indexed nested loops join can have a much lower cost than hash join in case the outer relation is small, and the index lookups fetch only a few tuples from the inner (indexed) relation. However, in case a secondary index is used, and the number of tuples in the outer relation is large, indexed nested loops join can have a very high cost, as compared to hash join. If the number of tuples in the outer relation is known at query optimization time, the best join algorithm can be chosen at that time. However, in some cases, for example, when there is a selection condition on the outer input, the optimizer makes a decision based on an estimate that may potentially be imprecise. The number of tuples in the outer relation may be found only at runtime, for example, after executing selection. Some systems allow a dynamic choice between the two algorithms at run time, after finding the number of tuples in the outer input.

15.5.5.5 Hybrid Hash Join

The **hybrid hash-join** algorithm performs another optimization; it is useful when memory sizes are relatively large but not all of the build relation fits in memory. The partitioning phase of the hash-join algorithm needs a minimum of one block of memory as a buffer for each partition that is created, and one block of memory as an input buffer. To reduce the impact of seeks, a larger number of blocks would be used as a buffer; let b_b denote the number of blocks used as a buffer for the input and for each partition. Hence, a total of $(n_h + 1) * b_b$ blocks of memory are needed for partitioning the two relations. If memory is larger than $(n_h + 1) * b_b$, we can use the rest of memory ($M - (n_h + 1) * b_b$ blocks) to buffer the first partition of the build input (i.e, s_0) so that it will not need to be written out and read back in. Further, the hash function is designed in such a way that the hash index on s_0 fits in $M - (n_h + 1) * b_b$ blocks, in order that, at the end of partitioning of s, s_0 is completely in memory and a hash index can be built on s_0.

When the system partitions r, it again does not write tuples in r_0 to disk; instead, as it generates them, the system uses them to probe the memory-resident hash index on s_0, and to generate output tuples of the join. After they are used for probing, the tuples can be discarded, so the partition r_0 does not occupy any memory space. Thus, a write and a read access have been saved for each block of both r_0 and s_0. The system writes out tuples in the other partitions as usual and joins them later. The savings of hybrid hash join can be significant if the build input is only slightly bigger than memory.

Telegram: @uni_k

www.konkur.in

If the size of the build relation is b_s, n_h is approximately equal to b_s/M. Thus, hybrid hash join is most useful if $M >> (b_s/M) * b_b$, or $M >> \sqrt{b_s * b_b}$, where the notation $>>$ denotes *much larger than*. For example, suppose the block size is 4 kilobytes, the build relation size is 5 gigabytes, and b_b is 20. Then, the hybrid hash-join algorithm is useful if the size of memory is significantly more than 20 megabytes; memory sizes of gigabytes or more are common on computers today. If we devote 1 gigabyte for the join algorithm, s_0 would be nearly 1 gigabyte, and hybrid hash join would be nearly 20 percent cheaper than hash join.

15.5.6 Complex Joins

Nested-loop and block nested-loop joins can be used regardless of the join conditions. The other join techniques are more efficient than the nested-loop join and its variants, but they can handle only simple join conditions, such as natural joins or equi-joins. We can implement joins with complex join conditions, such as conjunctions and disjunctions, by using the efficient join techniques, if we apply the techniques developed in Section 15.3.3 for handling complex selections.

Consider the following join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

One or more of the join techniques described earlier may be applicable for joins on the individual conditions $r \bowtie_{\theta_1} s$, $r \bowtie_{\theta_2} s$, $r \bowtie_{\theta_3} s$, and so on. We can compute the overall join by first computing the result of one of these simpler joins $r \bowtie_{\theta_i} s$; each pair of tuples in the intermediate result consists of one tuple from r and one from s. The result of the complete join consists of those tuples in the intermediate result that satisfy the remaining conditions:

$$\theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \cdots \wedge \theta_n$$

These conditions can be tested as tuples in $r \bowtie_{\theta_i} s$ are being generated.

A join whose condition is disjunctive can be computed in this way. Consider:

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$$

The join can be computed as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$$

Section 15.6 describes algorithms for computing the union of relations.

Telegram: @uni_k

www.konkur.in

15.5.7 Joins over Spatial Data

The join algorithms we have presented make no specific assumptions about the type of data being joined, but they do assume the use of standard comparison operations such as equality, less than, or greater than, where the values are linearly ordered.

Selection and join conditions on spatial data involve comparison operators that check if one region contains or overlaps another, or whether a region contains a particular point; and the regions may be multi-dimensional. Comparisons may pertain also to the distance between points, for example, finding a set of points closest to a given point in a two-dimensional space.

Merge-join cannot be used with such comparison operations, since there is no simple sort order over spatial data in two or more dimensions. Partitioning of data based on hashing is also not applicable, since there is no way to ensure that tuples that satisfy an overlap or containment predicate are hashed to the same value. Nested loops join can always be used regardless of the complexity of the conditions, but can be very inefficient on large datasets.

Indexed nested-loops join can however be used, if appropriate spatial indices are available. In Section 14.10, we saw several types of indices for spatial data, including R-trees, k-d trees, k-d-B trees, and quadtrees. Additional details on those indices appear in Section 24.4. These index structures enable efficient retrieval of spatial data based on predicates such as contains, contained in, or overlaps, and can also be effectively used to find nearest neighbors.

Most major database systems today incorporate support for indexing spatial data, and make use of them when processing queries using spatial comparison conditions.

15.6 Other Operations

Other relational operations and extended relational operations—such as duplicate elimination, projection, set operations, outer join, and aggregation—can be implemented as outlined in Section 15.6.1 through Section 15.6.5.

15.6.1 Duplicate Elimination

We can implement duplicate elimination easily by sorting. Identical tuples will appear adjacent to each other as a result of sorting, and all but one copy can be removed. With external sort–merge, duplicates found while a run is being created can be removed before the run is written to disk, thereby reducing the number of block transfers. The remaining duplicates can be eliminated during merging, and the final sorted run has no duplicates. The worst-case cost estimate for duplicate elimination is the same as the worst-case cost estimate for sorting of the relation.

We can also implement duplicate elimination by hashing, as in the hash-join algorithm. First, the relation is partitioned on the basis of a hash function on the whole tuple. Then, each partition is read in, and an in-memory hash index is constructed.

Telegram: @uni_k

www.konkur.in

While constructing the hash index, a tuple is inserted only if it is not already present. Otherwise, the tuple is discarded. After all tuples in the partition have been processed, the tuples in the hash index are written to the result. The cost estimate is the same as that for the cost of processing (partitioning and reading each partition) of the build relation in a hash join.

Because of the relatively high cost of duplicate elimination, SQL requires an explicit request by the user to remove duplicates; otherwise, the duplicates are retained.

15.6.2 Projection

We can implement projection easily by performing projection on each tuple, which gives a relation that could have duplicate records, and then removing duplicate records. Duplicates can be eliminated by the methods described in Section 15.6.1. If the attributes in the projection list include a key of the relation, no duplicates will exist; hence, duplicate elimination is not required. Generalized projection can be implemented in the same way as projection.

15.6.3 Set Operations

We can implement the *union*, *intersection*, and *set-difference* operations by first sorting both relations, and then scanning once through each of the sorted relations to produce the result. In $r \cup s$, when a concurrent scan of both relations reveals the same tuple in both files, only one of the tuples is retained. The result of $r \cap s$ will contain only those tuples that appear in both relations. We implement *set difference*, $r - s$, similarly, by retaining tuples in r only if they are absent in s.

For all these operations, only one scan of the two sorted input relations is required, so the cost is $b_r + b_s$ block transfers if the relations are sorted in the same order. Assuming a worst case of one block buffer for each relation, a total of $b_r + b_s$ disk seeks would be required in addition to $b_r + b_s$ block transfers. The number of seeks can be reduced by allocating extra buffer blocks.

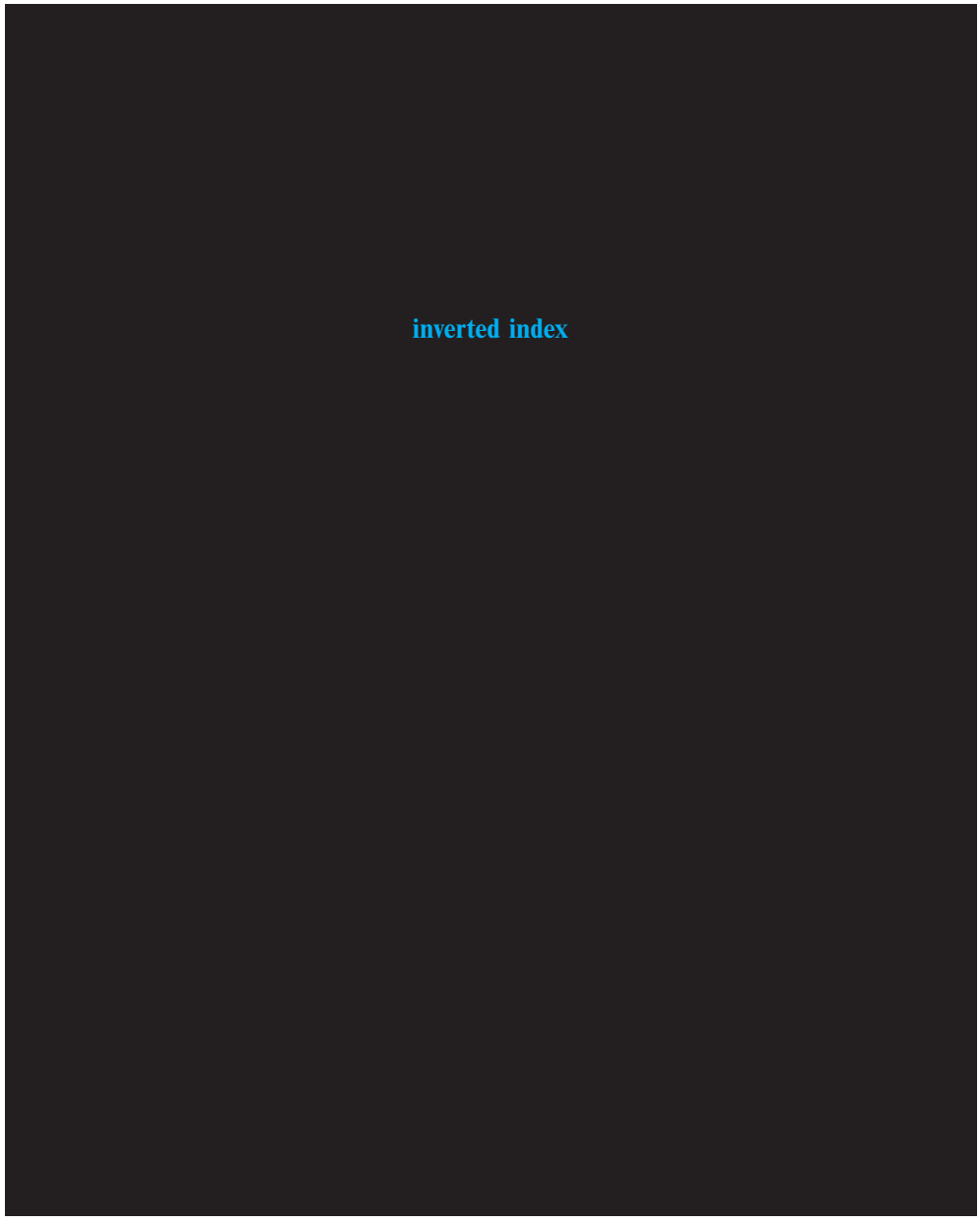
If the relations are not sorted initially, the cost of sorting has to be included. Any sort order can be used in the evaluation of set operations, provided that both inputs have that same sort order.

Hashing provides another way to implement these set operations. The first step in each case is to partition the two relations by the same hash function and thereby create the partitions $r_0, r_1, \ldots, r_{n_h}$ and $s_0, s_1, \ldots, s_{n_h}$. Depending on the operation, the system then takes these steps on each partition $i = 0, 1, \ldots, n_h$:

- $r \cup s$
  1. Build an in-memory hash index on r_i.
  2. Add the tuples in s_i to the hash index only if they are not already present.
  3. Add the tuples in the hash index to the result.

Telegram: @uni_k

www.konkur.in



- $r \cap s$
  1. Build an in-memory hash index on r_i.
  2. For each tuple in s_i, probe the hash index and output the tuple to the result only if it is already present in the hash index.

Telegram: @uni_k

www.konkur.in

- $r - s$

  1. Build an in-memory hash index on r_i.
  2. For each tuple in s_i, probe the hash index, and, if the tuple is present in the hash index, delete it from the hash index.
  3. Add the tuples remaining in the hash index to the result.

15.6.4 Outer Join

Recall the *outer-join operations* described in Section 4.1.3. For example, the natural left outer join *takes* ⟕ *student* contains the join of *takes* and *student*, and, in addition, for each *takes* tuple t that has no matching tuple in *student* (i.e, where *ID* is not in *student*), the following tuple t_1 is added to the result. For all attributes in the schema of *takes*, tuple t_1 has the same values as tuple t. The remaining attributes (from the schema of *student*) of tuple t_1 contain the value null.

We can implement the outer-join operations by using one of two strategies:

1. Compute the corresponding join, and then add further tuples to the join result to get the outer-join result. Consider the left outer-join operation and two relations: $r(R)$ and $s(S)$. To evaluate $r \mathbin{⟕}_\theta s$, we first compute $r \bowtie_\theta s$ and save that result as temporary relation q_1. Next, we compute $r - \Pi_R(q_1)$ to obtain those tuples in r that do not participate in the theta join. We can use any of the algorithms for computing the joins, projection, and set difference described earlier to compute the outer joins. We pad each of these tuples with null values for attributes from s, and add it to q_1 to get the result of the outer join.

   The right outer-join operation $r \mathbin{⟖}_\theta s$ is equivalent to $s \mathbin{⟕}_\theta r$ and can therefore be implemented in a symmetric fashion to the left outer join. We can implement the full outer-join operation $r \mathbin{⟗}_\theta s$ by computing the join $r \bowtie s$ and then adding the extra tuples of both the left and right outer-join operations, as before.

2. Modify the join algorithms. It is easy to extend the nested-loop join algorithms to compute the left outer join: Tuples in the outer relation that do not match any tuple in the inner relation are written to the output after being padded with null values. However, it is hard to extend the nested-loop join to compute the full outer join.

   Natural outer joins and outer joins with an equi-join condition can be computed by extensions of the merge-join and hash-join algorithms. Merge join can be extended to compute the full outer join as follows: When the merge of the two relations is being done, tuples in either relation that do not match any tuple in the other relation can be padded with nulls and written to the output. Similarly, we can extend merge join to compute the left and right outer joins by writing out nonmatching tuples (padded with nulls) from only one of the relations. Since the relations are sorted, it is easy to detect whether or not a tuple matches any tuples

Telegram: @uni_k

www.konkur.in

from the other relation. For example, when a merge join of *takes* and *student* is done, the tuples are read in sorted order of *ID*, and it is easy to check, for each tuple, whether there is a matching tuple in the other.

The cost estimates for implementing outer joins using the merge-join algorithm are the same as are those for the corresponding join. The only difference lies in the size of the result, and therefore in the block transfers for writing it out, which we did not count in our earlier cost estimates.

The extension of the hash-join algorithm to compute outer joins is left for you to do as an exercise (Exercise 15.21).

15.6.5 Aggregation

Recall the aggregation function (operator), discussed in Section 3.7. For example, the function

select *dept_name*, **avg** (*salary*)
from *instructor*
group by *dept_name*;

computes the average salary in each university department.

The aggregation operation can be implemented in the same way as duplicate elimination. We use either sorting or hashing, just as we did for duplicate elimination, but based on the grouping attributes (*dept_name* in the preceding example). However, instead of eliminating tuples with the same value for the grouping attribute, we gather them into groups and apply the aggregation operations on each group to get the result.

The cost estimate for implementing the aggregation operation is the same as the cost of duplicate elimination for aggregate functions such as **min**, **max**, **sum**, **count**, and **avg**.

Instead of gathering all the tuples in a group and then applying the aggregation operations, we can implement the aggregation operations **sum**, **min**, **max**, **count**, and **avg** on the fly as the groups are being constructed. For the case of **sum**, **min**, and **max**, when two tuples in the same group are found, the system replaces them with a single tuple containing the **sum**, **min**, or **max**, respectively, of the columns being aggregated. For the **count** operation, it maintains a running count for each group for which a tuple has been found. Finally, we implement the **avg** operation by computing the sum and the count values on the fly, and finally dividing the sum by the count to get the average.

If all tuples of the result fit in memory, the sort-based and the hash-based implementations do not need to write any tuples to disk. As the tuples are read in, they can be inserted in a sorted tree structure or in a hash index. When we use on-the-fly aggregation techniques, only one tuple needs to be stored for each of the groups. Hence, the sorted tree structure or hash index fits in memory, and the aggregation can be processed with just b_r block transfers (and 1 seek) instead of the $3b_r$ transfers (and a worst case of up to $2b_r$ seeks) that would be required

Telegram: @uni_k

www.konkur.in

15.7 Evaluation of Expressions

So far, we have studied how individual relational operations are carried out. Now we consider how to evaluate an expression containing multiple operations. The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk. An alternative approach is to evaluate several operations simultaneously in a **pipeline**, with the results of one operation passed on to the next, without the need to store a temporary relation.

In Section 15.7.1 and Section 15.7.2, we consider both the *materialization* approach and the *pipelining* approach. We shall see that the costs of these approaches can differ substantially, but also that there are cases where only the materialization approach is feasible.

15.7.1 Materialization

It is easiest to understand intuitively how to evaluate an expression by looking at a pictorial representation of the expression in an **operator tree**. Consider the expression:

$$\Pi_{name}(\sigma_{building = \text{“Watson”}}(department) \bowtie instructor)$$

in Figure 15.11.

If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation: the selection operation on *department*. The inputs to the lowest-level operations are relations in the database. We execute these operations using the algorithms that we studied earlier, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In our

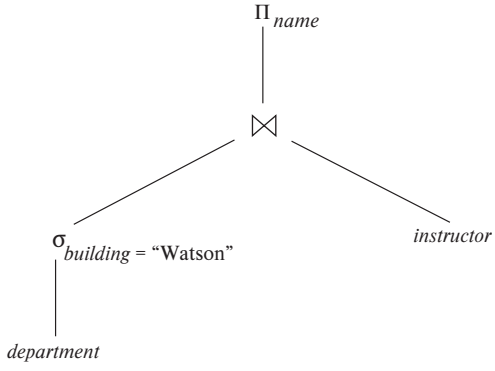


Figure 15.11 Pictorial representation of an expression.

Telegram: @uni_k

www.konkur.in

example, the inputs to the join are the *instructor* relation and the temporary relation created by the selection on *department*. The join can now be evaluated, creating another temporary relation.

By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In our example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

Evaluation as just described is called **materialized evaluation**, since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

The cost of a materialized evaluation is not simply the sum of the costs of the operations involved. When we computed the cost estimates of algorithms, we ignored the cost of writing the result of the operation to disk. To compute the cost of evaluating an expression as done here, we have to add the costs of all the operations, as well as the cost of writing the intermediate results to disk. We assume that the records of the result accumulate in a buffer, and, when the buffer is full, they are written to disk. The number of blocks written out, b_r, can be estimated as n_r/f_r, where n_r is the estimated number of tuples in the result relation r and f_r is the *blocking factor* of the result relation, that is, the number of records of r that will fit in a block. In addition to the transfer time, some disk seeks may be required, since the disk head may have moved between successive writes. The number of seeks can be estimated as $\lceil b_r/b_b \rceil$ where b_b is the size of the output buffer (measured in blocks).

Double buffering (using two buffers, with one continuing execution of the algorithm while the other is being written out) allows the algorithm to execute more quickly by performing CPU activity in parallel with I/O activity. The number of seeks can be reduced by allocating extra blocks to the output buffer and writing out multiple blocks at once.

15.7.2 Pipelining

We can improve query-evaluation efficiency by reducing the number of temporary files that are produced. We achieve this reduction by combining several relational operations into a *pipeline* of operations, in which the results of one operation are passed along to the next operation in the pipeline. Evaluation as just described is called **pipelined evaluation**.

For example, consider the expression $(\Pi_{a1,a2}(r \bowtie s))$. If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join and then reading back in the result to perform the projection. These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result and instead create the final result directly.

Telegram: @uni_k

www.konkur.in

Creating a pipeline of operations can provide two benefits:

1. It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation. Note that the cost formulae that we saw earlier for each operation included the cost of reading the result from disk. If the input to an operator o_i is pipelined from a preceding operator o_j, the cost of o_i should not include the cost of reading the input from disk; the cost formulae that we saw earlier can be modified accordingly.
2. It can start generating query results quickly, if the root operator of a query-evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

15.7.2.1 Implementation of Pipelining

We can implement a pipeline by constructing a single, complex operation that combines the operations that constitute the pipeline. Although this approach may be feasible for some frequently occurring situations, it is desirable in general to reuse the code for individual operations in the construction of a pipeline.

In the example of Figure 15.11, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Pipelines can be executed in either of two ways:

1. In a **demand-driven pipeline**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output and passes them up to its parent.
2. In a **producer-driven pipeline**, operations do not wait for requests to produce tuples, but instead generate the tuples **eagerly**. Each operation in a producer-driven pipeline is modeled as a separate process or thread within the system that takes a stream of tuples from its pipelined inputs and generates a stream of tuples for its output.

Telegram: @uni_k

www.konkur.in

We describe next how demand-driven and producer-driven pipelines can be implemented.

Each operation in a demand-driven pipeline can be implemented as an **iterator** that provides the following functions: *open*(), *next*(), and *close*(). After a call to *open*(), each call to *next*() returns the next output tuple of the operation. The implementation of the operation in turn calls *open*() and *next*() on its inputs, to get its input tuples when required. The function *close*() tells an iterator that no more tuples are required. The iterator maintains the **state** of its execution in between calls so that successive *next*() requests receive successive result tuples.

For example, for an iterator implementing the select operation using linear search, the *open*() operation starts a file scan, and the iterator's state records the point to which the file has been scanned. When the *next*() function is called, the file scan continues from after the previous point; when the next tuple satisfying the selection is found by scanning the file, the tuple is returned after storing the point where it was found in the iterator state. A merge-join iterator's *open*() operation would open its inputs, and if they are not already sorted, it would also sort the inputs. On calls to *next*(), it would return the next pair of matching tuples. The state information would consist of up to where each input had been scanned. Details of the implementation of iterators are left for you to complete in Practice Exercise 15.7.

Producer-driven pipelines, on the other hand, are implemented in a different manner. For each pair of adjacent operations in a producer-driven pipeline, the system creates a buffer to hold tuples being passed from one operation to the next. The processes or threads corresponding to different operations execute concurrently. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer so that the buffer has space for more tuples. At this point, the operation generates more tuples until the buffer is full again. The operation repeats this process until all the output tuples have been generated.

It is necessary for the system to switch between operations only when an output buffer is full or when an input buffer is empty and more input tuples are needed to generate any more output tuples. In a parallel-processing system, operations in a pipeline may be run concurrently on distinct processors (see Section 22.5.1).

Using producer-driven pipelining can be thought of as **pushing** data up an operation tree from below, whereas using demand-driven pipelining can be thought of as **pulling** data up an operation tree from the top. Whereas tuples are generated *eagerly* in producer-driven pipelining, they are generated **lazily**, on demand, in demand-driven pipelining. Demand-driven pipelining is used more commonly than producer-driven pipelining because it is easier to implement. However, producer-driven pipelining is very useful in parallel processing systems. Producer-driven pipelining has also been

Telegram: @uni_k

www.konkur.in

found to be more efficient than demand-driven pipelining on modern CPUs since it reduces the number of function call invocations as compared to demand-driven pipelining. Producer-driven pipelining is increasingly used in systems that generate machine code for high performance query evaluation.

15.7.2.2 Evaluation Algorithms for Pipelining

Query plans can be annotated to mark edges that are pipelined; such edges are called **pipelined edges**. In contrast, non-pipelined edges are referred to as **blocking edges** or **materialized edges**. The two operators connected by a pipelined edge must be executed concurrently, since one consumes tuples as the other generates them. Since a plan can have multiple pipelined edges, the set of all operators that are connected by pipelined edges must be executed concurrently. A query plan can be divided into subtrees such that each subtree has only pipelined edges, and the edges between the subtrees are non-pipelined. Each such subtree is called a **pipeline stage**. The query processor executes the plan one pipeline stage at a time, and concurrently executes all the operators in a single pipeline stage.

Some operations, such as sorting, are inherently **blocking operations**, that is, they may not be able to output any results until all tuples from their inputs have been examined.[7] But interestingly, blocking operators can consume tuples as they are generated, and can output tuples to their consumers as they are generated; such operations actually execute in two or more stages, and blocking actually happens between two stages of the operation.

For example, the external sort-merge operation actually has two steps: (i) run-generation, followed by (ii) merging. The run-generation step can accept tuples as they are generated by the input to the sort, and can thus be pipelined with the sort input. The merge step, on the other hand, can send tuples to its consumer as they are generated, and can thus be pipelined with the consumer of the sort operation. But the merge step can start only after the run-generation step has finished. We can thus model the sort-merge operator as two sub-operators connected to each other by a non-pipelined edge, but each of the sub-operators can be connected by pipelined edges to their input and output respectively.

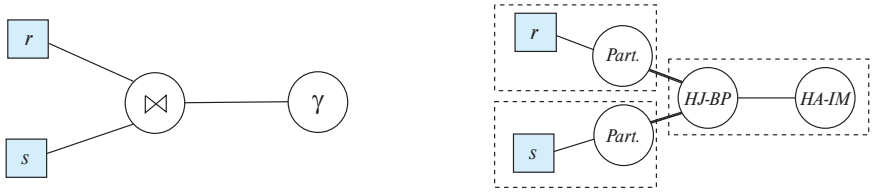
Other operations, such as join, are not inherently blocking, but specific evaluation algorithms may be blocking. For example, the indexed nested loops join algorithm can output result tuples as it gets tuples for the outer relation. It is therefore pipelined on its outer (left-hand side) relation; however, it is blocking on its indexed (right-hand side) input, since the index must be fully constructed before the indexed nested-loop join algorithm can execute.

The hash-join algorithm is a blocking operation on both inputs, since it requires both its inputs to be fully retrieved and partitioned before it outputs any tuples. How-

[7] Blocking operations such as sorting may be able to output tuples early if the input is known to satisfy some special properties such as being sorted, or partially sorted, already. However, in the absence of such information, blocking operations cannot output tuples early.

Telegram: @uni_k

www.konkur.in



(a) Logical Query (b) Pipelined Plan

Figure 15.12 Query plan with pipelining.

ever, hash-join partitions each of its inputs, and then performs multiple build-probe steps, once per partition. Thus, the hash-join algorithm has 3 steps: (i) partitioning of the first input, (ii) partitioning of the second input, and (iii) the build-probe step. The partitioning step for each input can accept tuples as they are generated by the input, and can thus be pipelined with its input. The build-probe step can output tuples to its consumer as the tuples are generated, and can thus be pipelined with its consumer. But the two partitioning steps are connected to the build-probe step by non-pipelined edges, since build-probe can start only after partitioning has been completed on both inputs.

Hybrid hash join can be viewed as partially pipelined on the probe relation, since it can output tuples from the first partition as tuples are received for the probe relation. However, tuples that are not in the first partition will be output only after the entire pipelined input relation is received. Hybrid hash join thus provides fully pipelined evaluation on its probe input if the build input fits entirely in memory, or nearly pipelined evaluation if most of the build input fits in memory.

Figure 15.12a shows a query that joins two relations r and s, and then performs an aggregation on the result; details of the join predicate, group by attributes and aggregation functions are omitted for simplicity. Figure 15.12b shows a pipelined plan for the query using hash join and in-memory hash aggregation. Pipelined edges are shown using a normal line, while blocking edges are shown using a bold line. Pipeline stages are enclosed in dashed boxes. Note that hash join has been split into three suboperators. Two of suboperators, shown abbreviated to *Part.*, partition r and s respectively. The third, abbreviated to *HJ-BP*, performs the build and probe phase of the hash join. The *HA-IM* operator is the in-memory hash aggregation operator. The edges from the partition operators to the *HJ-BP* operator are blocking edges, since the *HJ-BP* operator can start execution only after the partition operators have completed execution. The edges from the relations (assumed to be scanned using a relation scan operator) to the partition operators are pipelined, as is the edge from the *HJ-BP* operator to the *HA-IM* operator. The resultant pipeline stages are shown enclosed in dashed boxes.

In general, for each materialized edge we need to add the cost of writing the data to disk, and the cost of the consumer operator should include the cost of reading the data from disk. However, when a materialized edge is between suboperators of a single

Telegram: @uni_k

www.konkur.in


```
done_r := false;
done_s := false;
r := Ø;
s := Ø;
result := Ø;
while not done_r or not done_s do
   begin
      if queue is empty, then wait until queue is not empty;
      t := top entry in queue;
      if t = End_r then done_r := true
        else if t = End_s then done_s := true
           else if t is from input r
              then
                 begin
                    r := r ∪ {t};
                    result := result ∪ ({t} ⋈ s);
                 end
              else /* t is from input s */
                 begin
                    s := s ∪ {t};
                    result := result ∪ (r ⋈ {t});
                 end
   end
```

Figure 15.13 Double-pipelined join algorithm.

operator, for example between run generation and merge, the materialization cost has already been accounted for in the operators cost, and should not be added again.

In some applications, a join algorithm that is pipelined on both its inputs and its output is desirable. If both inputs are sorted on the join attribute, and the join condition is an equi-join, merge join can be used, with both its inputs and its output pipelined.

However, in the more common case that the two inputs that we desire to pipeline into the join are not already sorted, another alternative is the **double-pipelined join** technique, shown in Figure 15.13. The algorithm assumes that the input tuples for both input relations, r and s, are pipelined. Tuples made available for both relations are queued for processing in a single queue. Special queue entries, called End_r and End_s, which serve as end-of-file markers, are inserted in the queue after all tuples from r and s (respectively) have been generated. For efficient evaluation, appropriate indices should be built on the relations r and s. As tuples are added to r and s, the indices must be kept

Telegram: @uni_k

www.konkur.in

up to date. When hash indices are used on r and s, the resultant algorithm is called the **double-pipelined hash-join** technique.

The double-pipelined join algorithm in Figure 15.13 assumes that both inputs fit in memory. In case the two inputs are larger than memory, it is still possible to use the double-pipelined join technique as usual until available memory is full. When available memory becomes full, r and s tuples that have arrived up to that point can be treated as being in partition r_0 and s_0, respectively. Tuples for r and s that arrive subsequently are assigned to partitions r_1 and s_1, respectively, which are written to disk, and are not added to the in-memory index. However, tuples assigned to r_1 and s_1 are used to probe s_0 and r_0, respectively, before they are written to disk. Thus, the join of r_1 with s_0, and s_1 with r_0, is also carried out in a pipelined fashion. After r and s have been fully processed, the join of r_1 tuples with s_1 tuples must be carried out to complete the join; any of the join techniques we have seen earlier can be used to join r_1 with s_1.

15.7.3 Pipelines for Continuous-Stream Data

Pipelining is also applicable in situations where data are entered into the database in a continuous manner, as is the case, for example, for inputs from sensors that are continuously monitoring environmental data. Such data are called *data streams*, as we saw earlier in Section 10.5. Queries may be written over stream data in order to respond to data as they arrive. Such queries are called *continuous queries*.

The operations in a continuous query should be implemented using pipelined algorithms, so that results from the pipeline can be output without blocking. Producer-driven pipelines (which we discussed earlier in Section 15.7.2.1) are the best suited for continuous query evaluation.

Many such queries perform aggregation with windowing; tumbling windows which divide time into fixed size intervals, such as 1 minute, or 1 hour, are commonly used. Grouping and aggregation is performed separately on each window, as tuples are received; assuming memory size is large enough, an in-memory hash index is used to perform aggregation.

The result of aggregation on a window can be output once the system knows that no further tuples in that window will be received in future. If tuples are guaranteed to arrive sorted by timestamp, the arrival of a tuple of a following window indicates no more tuples will be received for an earlier window. If tuples may arrive out of order, streams must carry punctuations that indicate that all future tuples will have a timestamp greater than some specified value. The arrival of a punctuation allows the output of aggregates of windows whose end-timestamp is less than or equal to the timestamp specified by the punctuation.

15.8 Query Processing in Memory

The query processing algorithms that we have described so far focus on minimizing I/O cost. In this section, we discuss extensions to the query processing techniques that

Telegram: @uni_k

www.konkur.in

help minimize memory access costs by using cache-conscious query processing algorithms and query compilation. We then discuss query processing with column-oriented storage. The algorithms we describe in this section give significant benefits for memory resident data; they are also very useful with disk-resident data, since they can speed up processing once data has been brought into the in-memory buffer.

15.8.1 Cache-Conscious Algorithms

When data is resident in memory, access is much faster than if data were resident on magnetic disks, or even SSDs. However, it must be kept in mind that data already in CPU cache can be accessed as much as 100 times faster than data in memory. Modern CPUs have several levels of cache. Commonly used CPUs today have an L1 cache of size around 64 kilobytes, with a latency of about 1 nanosecond, an L2 cache of size around 256 kilobytes, with a latency of around 5 nanoseconds, and an L3 cache of having a size of around 10 megabytes, with a latency of 10 to 15 nanoseconds. In contrast, reading data in memory results in a latency of around 50 to 100 nanoseconds. For simplicity in the rest of this section we ignore the difference between the L1, L2 and L3 cache levels, and assume that there is only a single cache level.

As we saw in Section 14.4.7, the speed difference between cache memory and main memory, and the fact that data are transferred between main memory and cache in units of a *cache-line* (typically about 64 bytes), results in a situation where the relationship between cache and main memory is not dissimilar to the relationship between main memory and disk (although with smaller speed differences). But there is a difference: while the contents of the main memory buffers disk-based data are controlled by the database system, CPU cache is controlled by the algorithms built into the computer hardware. Thus, the database system cannot directly control what is kept in cache.

However, query processing algorithms can be designed in a way that the makes the best use of cache, to optimize performance. Here are some ways this can be done:

- To sort a relation that is in-memory, we use the external merge-sort algorithm, with the run size chosen such that the run fits into the cache; assuming we focus on the L3 cache, each run should be a few megabytes in size. We then use an in-memory sorting algorithm on each run; since the run fits in cache, cache misses are likely to be minimal when the run is sorted. The sorted runs (all of which are in memory) are then merged. Merging is cache efficient, since access to the runs is sequential: when a particular word is accessed from memory, the cache line that is fetched will contain the words that would be accessed next from that run.

  To sort a relation larger than memory, we can use external sort-merge with much larger run sizes, but use the in-memory merge-sort technique we just described to perform the in-memory sort of the large runs.

- Hash-join requires probing of an index on the build relation. If the build relation fits in memory, an index could be built on the whole relation; however, cache hits during probe can be maximized by partitioning the relations into smaller pieces

Telegram: @uni_k

www.konkur.in

such that each partition of the build-relation along with the index fits in the cache. Each partition is processed separately, with a build and a probe phase; since the build partition and its index fit in cache, cache misses are minimized during the build as well as the probe phase.

For relations larger than memory, the first stage of hash-join should partition the two relations such that for each partition, the partitions of the two relations together fit in memory. The technique just described can then be used to perform the hash join on each of these partitions, after fetching the contents into memory.

- Attributes in a tuple can be arranged such that attributes that tend to be accessed together are laid out consecutively. For example, if a relation is often used for aggregation, those attributes used as group by attributes, and those that are aggregated upon, can be stored consecutively. As a result, if there is a cache miss on one attribute, the cache line that is fetched would contain attributes that are likely to be used immediately.

Cache-aware algorithms are of increasing importance in modern database systems, since memory sizes are often large enough that much of the data is memory-resident.

In cases where the requisite data item is not in cache, there is a processing **stall** while the data item is retrieved from memory and loaded into cache. In order to continue to make use of the core that made the request resulting in the stall, the operating system maintains multiple threads of execution on which a core may work. Parallel query processing algorithms, which we study in Chapter 22 can use multiple threads running on a single CPU core; if one thread is stalled, another can start execution so the CPU core is utilized better.

15.8.2 Query Compilation

With data resident in memory, CPU cost becomes the bottleneck, and minimizing CPU cost can give significant benefits. Traditional databases query processors act as interpreters that execute a query plan. However, there is a significant overhead due to interpretation: for example, to access an attribute of a record, the query execution engine may repeatedly look up the relation meta-data to find the offset of the attribute within the record, since the same code must work for all relations. There is also significant overhead due to function calls that are performed for each record processed by an operation.

To avoid overhead due to interpretation, modern main-memory databases compile query plans into machine code or intermediate level byte-code. For example, the compiler can compute the offset of an attribute at compile time, and generate code where the offset is a constant. The compiler can also combine the code for multiple functions in a way that minimizes function calls. With these, and other related optimizations, compiled code has been found to execute faster, by up to a factor of 10, than interpreted code.

Telegram: @uni_k

www.konkur.in

15.8.3 Column-Oriented Storage

In Section 13.6, we saw that in data-analytic applications, only a few attributes of a large schema may be needed, and that in such cases, storing a relation by column instead of by row may be advantageous. Selection operations on a single attribute (or small number of attributes) have significantly lower cost in a column store since only the relevant attributes need to be accessed. However, since accessing each attribute requires its own data access, the cost of retrieving many attributes is higher and may incur additional seeks if data are stored on disk.

Because column stores permit efficient access to many values for a given attribute at once, they are well suited to exploit the vector-processing capabilities of modern processors. This capability allows certain operations (such as comparisons and aggregations) to be performed in a parallel on multiple attribute values. When compiling query plans to machine code, the compiler can generate vector-processing instructions supported by the processor.

15.9 Summary

- The first action that the system must perform on a query is to translate the query into its internal form, which (for relational database systems) is usually based on the relational algebra. In the process of generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of relations in the database, and so on. If the query was expressed in terms of a view, the parser replaces all references to the view name with the relational-algebra expression to compute the view.
- Given a query, there are generally a variety of methods for computing the answer. It is the responsibility of the query optimizer to transform the query as entered by the user into an equivalent query that can be computed more efficiently. Chapter 16 covers query optimization.
- We can process simple selection operations by performing a linear scan or by making use of indices. We can handle complex selections by computing unions and intersections of the results of simple selections.
- We can sort relations larger than memory by the external sort – merge algorithm.
- Queries involving a natural join may be processed in several ways, depending on the availability of indices and the form of physical storage for the relations.
    - If the join result is almost as large as the Cartesian product of the two relations, a *block nested-loop* join strategy may be advantageous.
    - If indices are available, the *indexed nested-loop* join can be used.

Telegram: @uni_k

www.konkur.in

  - If the relations are sorted, a *merge join* may be desirable. It may be advantageous to sort a relation prior to join computation (so as to allow use of the merge-join strategy).
  - The *hash-join* algorithm partitions the relations into several pieces, such that each piece of one of the relations fits in memory. The partitioning is carried out with a hash function on the join attributes so that corresponding pairs of partitions can be joined independently.
- Duplicate elimination, projection, set operations (union, intersection, and difference), and aggregation can be done by sorting or by hashing.
- Outer-join operations can be implemented by simple extensions of join algorithms.
- Hashing and sorting are dual, in the sense that any operation such as duplicate elimination, projection, aggregation, join, and outer join that can be implemented by hashing can also be implemented by sorting, and vice versa; that is, any operation that can be implemented by sorting can also be implemented by hashing.
- An expression can be evaluated by means of materialization, where the system computes the result of each subexpression and stores it on disk and then uses it to compute the result of the parent expression.
- Pipelining helps to avoid writing the results of many subexpressions to disk by using the results in the parent expression even as they are being generated.

Review Terms

- Query processing
- Evaluation primitive
- Query-execution plan
- Query-evaluation plan
- Query-execution engine
- Measures of query cost
- Sequential I/O
- Random I/O
- File scan
- Linear search
- Selections using indices
- Access paths
- Index scans
- Conjunctive selection
- Disjunctive selection
- Composite index
- Intersection of identifiers
- External sorting
- External sort–merge
- Runs
- *N*-way merge
- Equi-join
- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge join
- Sort-merge join
- Hybrid merge join

Telegram: @uni_k

www.konkur.in

- Hash-join
  - Build
  - Probe
  - Build input
  - Probe input
  - Recursive partitioning
  - Hash-table overflow
  - Skew
  - Fudge factor
  - Overflow resolution
  - Overflow avoidance
- Hybrid hash-join
- Spatial join
- Operator tree
- Materialized evaluation
- Double buffering
- Pipelined evaluation
  - Demand-driven pipeline (lazy, pulling)
  - Producer-driven pipeline (eager, pushing)
  - Iterator
  - Pipeline stages
- Double-pipelined join
- Continuous query evaluation

Practice Exercises

15.1 Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most three blocks. Show the runs created on each pass of the sort-merge algorithm when applied to sort the following tuples on the first attribute: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12).

15.2 Consider the bank database of Figure 15.14, where the primary keys are underlined, and the following SQL query:

select *T.branch_name*
from *branch T*, *branch S*
where *T.assets* > *S.assets* **and** *S.branch_city* = "Brooklyn"

Write an efficient relational-algebra expression that is equivalent to this query. Justify your choice.

15.3 Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 20,000 tuples, r_2 has 45,000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Estimate the number of block transfers and seeks required using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join.

b. Block nested-loop join.

Telegram: @uni_k

www.konkur.in

c. Merge join.

d. Hash join.

15.4 The indexed nested-loop join algorithm described in Section 15.5.3 can be inefficient if the index is a secondary index and there are multiple tuples with the same value for the join attributes. Why is it inefficient? Describe a way, using sorting, to reduce the cost of retrieving tuples of the inner relation. Under what conditions would this algorithm be more efficient than hybrid merge join?

15.5 Let r and s be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest-cost way (in terms of I/O operations) to compute $r \bowtie s$? What is the amount of memory required for this algorithm?

15.6 Consider the bank database of Figure 15.14, where the primary keys are underlined. Suppose that a B^+-tree index on *branch_city* is available on relation *branch*, and that no other index is available. List different ways to handle the following selections that involve negation:

a. $\sigma_{\neg(branch_city < \text{“Brooklyn”})}(branch)$

b. $\sigma_{\neg(branch_city = \text{“Brooklyn”})}(branch)$

c. $\sigma_{\neg(branch_city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

15.7 Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Your pseudocode must define the standard iterator functions *open*( ), *next*( ), and *close*( ). Show what state information the iterator must maintain between calls.

15.8 Design sort-based and hash-based algorithms for computing the relational division operation (see Practice Exercise 2.9 for a definition of the division operation).

---

branch(<u>*branch_name*</u>, *branch_city, assets*)
customer (<u>*customer_name*</u>, *customer_street, customer_city*)
loan (<u>*loan_number*</u>, *branch_name, amount*)
borrower (<u>*customer_name*</u>, <u>*loan_number*</u>)
account (<u>*account_number*</u>, *branch_name, balance*)
depositor (<u>*customer_name*</u>, <u>*account_number*</u>)

Figure 15.14 Bank database.

Telegram: @uni_k

www.konkur.in

15.9 What is the effect on the cost of merging runs if the number of buffer blocks per run is increased while overall memory available for buffering runs remains fixed?

15.10 Consider the following extended relational-algebra operators. Describe how to implement each operation using sorting and using hashing.

a. **Semijoin** ($\ltimes_\theta$): The multiset semijoin operator $r \ltimes_\theta s$ is defined as follows: if a tuple r_i appears n times in r, it appears n times in the result of $r \ltimes_\theta$ if there is at least one tuple s_j such that r_i and s_j satisfy predicate θ; otherwise r_i does not appear in the result.

b. **Anti-semijoin** ($\overline{\ltimes}_\theta$): The multiset anti-semijoin operator $r \overline{\ltimes}_\theta s$ is defined as follows: if a tuple r_i appears n times in r, it appears n times in the result of $r \ltimes_\theta$ if there does not exist any tuple s_j in s such that r_i and s_j satisfy predicate θ; otherwise r_i does not appear in the result.

15.11 Suppose a query retrieves only the first K results of an operation and terminates after that. Which choice of demand-driven or producer-driven pipelining (with buffering) would be a good choice for such a query? Explain your answer.

15.12 Current generation CPUs include an *instruction cache*, which caches recently used instructions. A function call then has a significant overhead because the set of instructions being executed changes, resulting in cache misses on the instruction cache.

a. Explain why producer-driven pipelining with buffering is likely to result in a better instruction cache hit rate, as compared to demand-driven pipelining.

b. Explain why modifying demand-driven pipelining by generating multiple results on one call to *next*(), and returning them together, can improve the instruction cache hit rate.

15.13 Suppose you want to find documents that contain at least k of a given set of n keywords. Suppose also you have a keyword index that gives you a (sorted) list of identifiers of documents that contain a specified keyword. Give an efficient algorithm to find the desired set of documents.

15.14 Suggest how a document containing a word (such as "leopard") can be indexed such that it is efficiently retrieved by queries using a more general concept (such as "carnivore" or "mammal"). You can assume that the concept hierarchy is not very deep, so each concept has only a few generalizations (a concept can, however, have a large number of specializations). You can also assume that you are provided with a function that returns the concept for each word in a document. Also suggest how a query using a specialized concept can retrieve documents using a more general concept.

Telegram: @uni_k

www.konkur.in

15.15 Explain why the nested-loops join algorithm (see Section 15.5.1) would work poorly on a database stored in a column-oriented manner. Describe an alternative algorithm that would work better, and explain why your solution is better.

15.16 Consider the following queries. For each query, indicate if column-oriented storage is likely to be beneficial or not, and explain why.

a. Fetch ID, *name* and *dept_name* of the student with ID 12345.

b. Group the *takes* relation by *year* and *course_id*, and find the total number of students for each (*year*, *course_id*) combination.

Exercises

15.17 Suppose you need to sort a relation of 40 gigabytes, with 4-kilobyte blocks, using a memory size of 40 megabytes. Suppose the cost of a seek is 5 milliseconds, while the disk transfer rate is 40 megabytes per second.

a. Find the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$.

b. In each case, how many merge passes are required?

c. Suppose a flash storage device is used instead of a disk, and it has a latency of 20 microsecond and a transfer rate of 400 megabytes per second. Recompute the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$, in this setting.

15.18 Why is it not desirable to force users to make an explicit choice of a query-processing strategy? Are there cases in which it *is* desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

15.19 Design a variant of the hybrid merge-join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

15.20 Estimate the number of block transfers and seeks required by your solution to Exercise 15.19 for $r_1 \bowtie r_2$, where r_1 and r_2 are as defined in Exercise 15.3.

15.21 The hash-join algorithm as described in Section 15.5.5 computes the natural join of two relations. Describe how to extend the hash-join algorithm to compute the natural left outer join, the natural right outer join, and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the *takes* and *student* relations.

Telegram: @uni_k

www.konkur.in

15.22 Suppose you have to compute ${}_{A}\gamma_{sum(C)}(r)$ as well as ${}_{A,B}\gamma_{sum(C)}(r)$. Describe how to compute these together using a single sorting of r.

15.23 Write pseudocode for an iterator that implements a version of the sort – merge algorithm where the result of the final merge is pipelined to its consumers. Your pseudocode must define the standard iterator functions *open*(), *next*(), and *close*(). Show what state information the iterator must maintain between calls.

15.24 Explain how to split the hybrid hash-join operator into sub-operators to model pipelining. Also explain how this split is different from the split for a hash-join operator.

15.25 Suppose you need to sort relation r using sort–merge and merge–join the result with an already sorted relation s.

a. Describe how the sort operator is broken into suboperators to model the pipelining in this case.

b. The same idea is applicable even if both inputs to the merge join are the outputs of sort–merge operations. However, the available memory has to be shared between the two merge operations (the merge–join algorithm itself needs very little memory). What is the effect of having to share memory on the cost of each sort-merge operation?

Further Reading

[Graefe (1993)] presents an excellent survey of query-evaluation techniques. [Faerber et al. (2017)] describe main-memory database implementation techniques, including query processing techniques for main-memory databases, while [Kemper et al. (2012)] describes techniques for query processing with in-memory columnar data. [Samet (2006)] provides a textbook description of spatial data structures, while [Shekhar and Chawla (2003)] provides a textbook description of spatial databases, including indexing and query processing techniques. Textbook descriptions of techniques for indexing documents, and efficiently computing ranked answers to keyword queries may be found in [Manning et al. (2008)].

Bibliography

[Faerber et al. (2017)] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo, "Main Memory Database Systems", *Foundations and Trends in Databases*, Volume 8, Number 1-2 (2017), pages 1–130.

[Graefe (1993)] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, Volume 25, Number 2 (1993).

Telegram: @uni_k

www.konkur.in

[Kemper et al. (2012)] A. Kemper, T. Neumann, F. Funke, V. Leis, and H. Mühe, "HyPer: Adapting Columnar Main-Memory Data Management for Transaction AND Query Processing", *IEEE Data Engineering Bulletin*, Volume 35, Number 1 (2012), pages 46–51.

[Manning et al. (2008)] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press (2008).

[Samet (2006)] H. Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann (2006).

[Shekhar and Chawla (2003)] S. Shekhar and S. Chawla, *Spatial Databases: A TOUR*, Pearson (2003).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 16



Query Optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play.

One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute. Another aspect is selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often substantial and may be several orders of magnitude. Hence, it is worthwhile for the system to spend a substantial amount of time on the selection of a good strategy for processing a query, even if the query is executed only once.

16.1 Overview

Consider the following relational-algebra expression, for the query "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."[1]

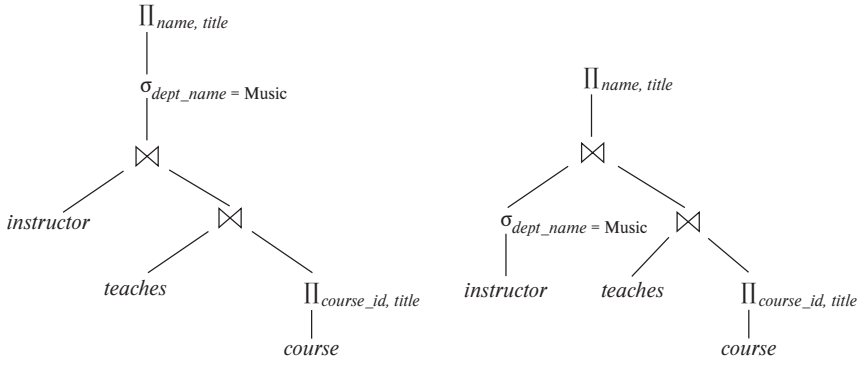
$$\Pi_{name,title}\,(\sigma_{dept_name = \text{"Music"}}\,(instructor \bowtie (teaches \bowtie \Pi_{course_id,title}(course))))$$

The subexpression $instructor \bowtie teaches \bowtie \Pi_{course_id,title}(course)$ in the preceding expression can create a very large intermediate result. However, we are interested in only a few tuples of this intermediate result, namely, those pertaining to instructors in the

[1]Note that the projection of *course* on (*course_id*, *title*) is required since *course* shares an attribute *dept_name* with *instructor*; if we did not remove this attribute using the projection, the above expression using natural joins would return only courses from the Music department, even if some Music department instructors taught courses in other departments.

Telegram: @uni_k

www.konkur.in



(a) Initial expression tree (b) Transformed expression tree

Figure 16.1 Equivalent expressions.

Music department, and in only two of the nine attributes of this relation. Since we are concerned with only those tuples in the *instructor* relation that pertain to the Music department, we do not need to consider those tuples that do not have *dept_name* = "Music". By reducing the number of tuples of the *instructor* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression:

$$\Pi_{name,title}\left((\sigma_{dept_name = \text{"Music"}}(instructor)) \bowtie (teaches \bowtie \Pi_{course_id,title}(course))\right)$$

which is equivalent to our original algebra expression, but which generates smaller intermediate relations. Figure 16.1 depicts the initial and transformed expressions.

An evaluation plan defines exactly what algorithm should be used for each operation and how the execution of the operations should be coordinated. Figure 16.2 illustrates one possible evaluation plan for the expression from Figure 16.1(b). As we have seen, several different algorithms can be used for each relational operation, giving rise to alternative evaluation plans. In the figure, hash join has been chosen for one of the join operations, while the other uses merge join, after sorting the relations on the join attribute, which is ID. All edges are assumed to be pipelined, unless marked as materialized. With pipelined edges the output of the producer is sent directly to the consumer, without being written out to disk; with materialized edges, on the other hand, the output is written to disk, and then read from the disk by the consumer. There are no materialized edges in the evaluation plan in Figure 16.2, although some of the operators, such as sort and hash join, can be represented using suboperators with materialized edges between the suboperators, as we saw in Section 15.7.2.2.

Given a relational-algebra expression, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least costly way of generating the result (or, at least, is not much costlier than the least costly way).

Telegram: @uni_k

www.konkur.in

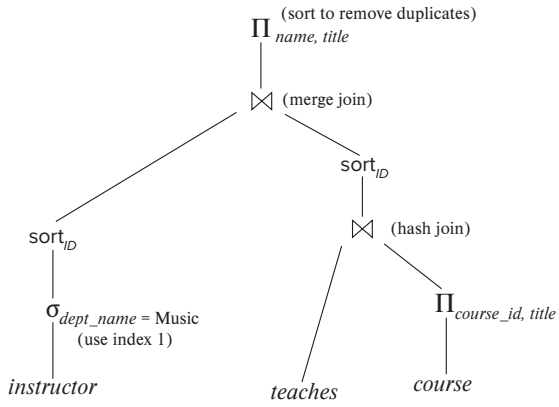


Figure 16.2 An evaluation plan.

The expression that we saw in Figure 16.1 may not necessarily lead to the least-cost evaluation plan for computing the result, since it still computes the join of the entire *teaches* relation with the *course* relation. The following expression gives the same final result, but generates smaller intermediate results, since it joins *teaches* with only *instructor* tuples corresponding to the Music department, and then joins that result with *course*.

$$\Pi_{name,title}\left((\sigma_{dept_name = \text{“Music”}}(instructor)) \bowtie teaches\right) \bowtie \Pi_{course_id,title}(course))$$

Regardless of the way the query is written, it is the job of the optimizer to find the least-cost plan for the query.

To find the least costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression and to choose the least costly one. Generation of query-evaluation plans involves three steps: (1) generating expressions that are logically equivalent to the given expression, (2) annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans, and (3) estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least.

Steps (1), (2), and (3) are interleaved in the query optimizer—some expressions are generated and annotated to generate evaluation plans, then further expressions are generated and annotated, and so on. As evaluation plans are generated, their costs are estimated by using statistical information about the relations, such as relation sizes and index depths.

To implement the first step, the query optimizer must generate expressions equivalent to a given expression. It does so by means of *equivalence rules* that specify how to transform an expression into a logically equivalent one. We describe these rules in Section 16.2.

In Section 16.3 we describe how to estimate statistics of the results of each operation in a query plan. Using these statistics with the cost formulae in Chapter 15 allows

Telegram: @uni_k

www.konkur.in



us to estimate the costs of individual operations. The individual costs are combined to determine the estimated cost of evaluating a given relational-algebra expression, as outlined in Section 15.7.

In Section 16.4, we describe how to choose a query-evaluation plan. We can choose one based on the estimated cost of the plans. Since the cost is an estimate, the selected plan is not necessarily the least costly plan; however, as long as the estimates are good, the plan is likely to be the least costly one, or not much more costly than it.

Telegram: @uni_k

www.konkur.in

Finally, materialized views help to speed up processing of certain queries. In Section 16.5, we study how to "maintain" materialized views—that is, to keep them up-to-date—and how to perform query optimization with materialized views.

16.2 Transformation of Relational Expressions

A query can be expressed in several different ways, with different costs of evaluation. In this section, rather than take the relational expression as given, we consider alternative, equivalent expressions.

Two relational-algebra expressions are said to be **equivalent** if, on every legal database instance, the two expressions generate the same set of tuples. (Recall that a legal database instance is one that satisfies all the integrity constraints specified in the database schema.) Note that the order of the tuples is irrelevant; the two expressions may generate the tuples in different orders, but would be considered equivalent as long as the set of tuples is the same.

In SQL, the inputs and outputs are multisets of tuples, and the multiset version of the relational algebra (described in Note 3.1 on page 80, Note 3.2 on page 97, and Note 3.3 on page 108) is used for evaluating SQL queries. Two expressions in the *multiset* version of the relational algebra are said to be equivalent if on every legal database the two expressions generate the same multiset of tuples. The discussion in this chapter is based on the relational algebra. We leave extensions to the multiset version of the relational algebra to you as exercises.

16.2.1 Equivalence Rules

An **equivalence rule** says that expressions of two forms are equivalent. We can replace an expression of the first form with an expression of the second form, or vice versa—that is, we can replace an expression of the second form by an expression of the first form—since the two expressions generate the same result on any valid database. The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.

We now describe several equivalence rules on relational-algebra expressions. Some of the equivalences listed appear in Figure 16.3. We use $\theta, \theta_1, \theta_2$, and so on to denote predicates, L_1, L_2, L_3, and so on to denote lists of attributes, and E, E_1, E_2, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression and can be used wherever E appears.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are **commutative**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma \quad (\sigma \quad (E))$$

Telegram: @uni_k

www.konkur.in

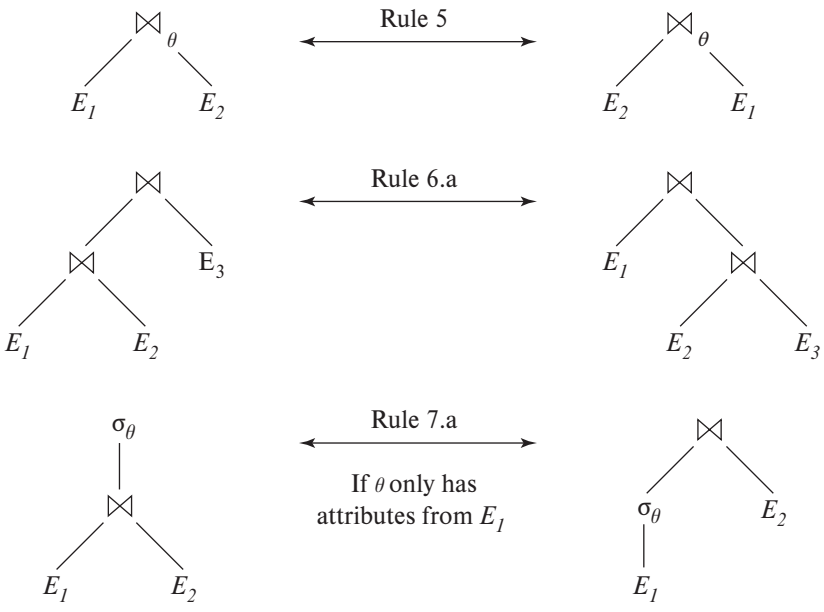


Figure 16.3 Pictorial representation of equivalences.

3. Only the final operations in a sequence of projection operations are needed; the others can be omitted. This transformation can also be referred to as a cascade of Π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$$

where $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$.

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_\theta(E_1 \times E_2) \equiv E_1 \bowtie_\theta E_2$

      This expression is just the definition of the theta join.

   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta-join operations are commutative.

$$E_1 \bowtie_\theta E_2 \equiv E_2 \bowtie_\theta E_1$$

Recall that the natural-join operator is simply a special case of the theta-join operator; hence, natural joins are also commutative.

The order of attributes differs between the left-hand side and right-hand side of the commutativity rule, so the equivalence does not hold if the order of attributes is taken into account. Since we use a version of relational algebra where every attribute must have a name for it to be referenced, the order of attributes

Telegram: @uni_k

www.konkur.in

does not actually matter, except when the result is finally displayed. When the order does matter, a projection operation can be added to one of the sides of the equivalence to appropriately reorder attributes. However, for simplicity, we omit the projection and ignore the attribute order in all our equivalence rules.

6. a. Natural-join operations are **associative**.

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

b. Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3. Any of these conditions may be empty; hence, it follows that the Cartesian product ($\times$) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

7. The selection operation distributes over the theta-join operation under the following two conditions:

a. Selection distributes over the theta-join operation when all the attributes in selection condition θ_1 involve only the attributes of one of the expressions (say, E_1) being joined.

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} E_2$$

b. Selection distributes over the theta-join operation when selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta-join operation under the following conditions.

a. Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) \equiv (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

b. Consider a join $E_1 \bowtie_{\theta} E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in L_1 and let L_4 be attributes of E_2 that are involved in join condition θ, but are not in L_2. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) \equiv \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

Similar equivalences hold for outer join operations $\overset{\frown}{\bowtie}$, $\overset{\frown}{\bowtie}$ and $\overset{\frown}{\bowtie}$.

Telegram: @uni_k

www.konkur.in

9. The set operations union and intersection are commutative.

   a. $E_1 \cup E_2 \equiv E_2 \cup E_1$

   b. $E_1 \cap E_2 \equiv E_2 \cap E_1$

   Set difference is not commutative.

10. Set union and intersection are associative.

    a. $(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$

    b. $(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$

11. The selection operation distributes over the union, intersection, and set-difference operations.

    a. $\sigma_\theta(E_1 \cup E_2) \equiv \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$

    b. $\sigma_\theta(E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$

    c. $\sigma_\theta(E_1 - E_2) \equiv \sigma_\theta(E_1) - \sigma_\theta(E_2)$

    d. $\sigma_\theta(E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$

    e. $\sigma_\theta(E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

    The preceding equivalence does not hold if $-$ is replaced by $\cup$.

12. The projection operation distributes over the union operation

$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

    provided E_1 and E_2 have the same schema.

13. Selection distributes over aggregation under the following conditions. Let G be a set of group by attributes, and A a set of aggregate expressions. When θ only involves attributes in G, the following equivalence holds.

$$\sigma_\theta({}_G\gamma_A(E) \equiv {}_G\gamma_A(\sigma_\theta(E))$$

14. a. Full outer join is commutative.

$$E_1 \mathbin{⟗} E_2 \equiv E_2 \mathbin{⟗} E_2$$

    b. Left and right outer join are not commutative. However, left outer join and right outer join can be exchanged as follows.

$$E_1 ⟕ E_2 \equiv E_2 ⟖ E_1$$

15. Selection distributes over left and right outer join under some conditions. Specifically, when the selection condition θ_1 involves only the attributes of one of the expressions being joined, say E_1, the following equivalences hold.

Telegram: @uni_k

www.konkur.in

a. $\sigma_{\theta_1}(E_1 \rtimes_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \rtimes_\theta E_2$

b. $\sigma_{\theta_1}(E_2 \ltimes_\theta E_1) \equiv (E_2 \ltimes_\theta (\sigma_{\theta_1}(E_1)))$

16. Outer joins can be replaced by inner joins under some conditions. Specifically, if θ_1 has the property that it evaluates to false or unknown whenever the attributes of E_2 are null, then the following equivalences hold.

a. $\sigma_{\theta_1}(E_1 \rtimes_\theta E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2)$

b. $\sigma_{\theta_1}(E_2 \ltimes_\theta E_1) \equiv \sigma_{\theta_1}(E_2 \bowtie_\theta E_2)$

A predicate θ_1 satisfying the above property is said to be **null rejecting** on E_2. For example, if θ_1 is of the form $A < 4$ where A is an attribute from E_2, then θ_1 would evaluate to unknown whenever A is null, and as a result any tuples in $E_1 \rtimes_\theta E_2$ that are not in $E_1 \bowtie_\theta E_2$ would be rejected by σ_{θ_1}. We can therefore replace the outer join by an inner join (or vice versa).

More generally, the condition would hold if θ_1 is of the form $\theta_1^1 \wedge \theta_1^2 \wedge \ldots \wedge \theta_1^k$, and at least one of the terms θ_1^i is of the form e_1 *relop* e_2, where e_1 and e_2 are arithmetic or string expressions involving at least one attribute from E_2, and *relop* is any of $<, \leq, =, \geq, >$.

This is only a partial list of equivalences. More equivalences are discussed in the exercises.

Some equivalences that hold for joins do not hold for outer joins. For example, the selection operation does not distribute over outer join when the conditions specified in rule 15.a or rule 15.b do hold. To see this, we look at the expression:

$$\sigma_{year=2017}(instructor \rtimes teaches)$$

and consider the case of an instructor who teaches no courses at all, regardless of year. In the above expression, the left outer join retains a tuple for each such instructor with a null value for *year*. Then the selection operation removes those tuples since the predicate *null*=2017 evaluates to *unknown*, and such instructors do not appear in the result. However, if we push the selection operation down to *teaches*, the resulting expression:

$$instructor \rtimes \sigma_{year=2017}(teaches)$$

is syntactically correct since the selection predicate includes only attributes from *teaches*, but the result is different. For an instructor that does not teach at all, the *instructor* tuple appears in the result of $instructor \rtimes \sigma_{year=2017}(teaches)$, but not in the result of $\sigma_{year=2017}(instructor \rtimes teaches)$. The following equivalence, however, does hold:

$$\sigma_{year=2017}(instructor \rtimes teaches) \equiv \sigma_{year=2017}(instructor \bowtie teaches)$$

As another example, unlike inner joins, outer joins are not associative. We show thus using an example for the natural left outer join. Similar examples can be con-

Telegram: @uni_k

www.konkur.in

structed for natural right and natural full outer join, as well as for the corresponding theta-join versions of the outer join operations.

Let relation $r(A, B)$ be a relation consisting of the single tuple $(1, 1)$, $s(B, C)$ be a relation consisting of the single tuple $(1, 1)$, and $t(A, C)$ be an empty relation with no tuples. We shall show that for this example,

$$(r \rtimes s) \rtimes t \not\equiv r \rtimes (s \rtimes t)$$

To see this, note first that $(r \rtimes s)$ produces a result with schema (A, B, C) having one tuple $(1, 1, 1)$. Computing the left outer join of that result with relation t produces a result with schema (A, B, C) having one tuple $(1, 1, 1)$. Next, we look at the expression $r \rtimes (s \rtimes t)$, and note that $s \rtimes t$ produces a result with schema (A, B, C) having one tuple $(null, 1, 1)$. Computing the left outer join of r with that result produces a result with schema (A, B, C) having one tuple $(1, 1, null)$.

16.2.2 Examples of Transformations

We now illustrate the use of the equivalence rules. We use our university example with the relation schemas:

instructor(*ID*, *name*, *dept_name*, *salary*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
course(*course_id*, *title*, *dept_name*, *credits*)

In our example in Section 16.1, the expression:

$$\Pi_{name,title}\,(\sigma_{dept_name = \text{“Music”}}\,(instructor \bowtie (teaches \bowtie \Pi_{course_id,title}(course))))$$

was transformed into the following expression:

$$\Pi_{name,title}\,((\sigma_{dept_name = \text{“Music”}}\,(instructor)) \bowtie (teaches \bowtie \Pi_{course_id,title}(course)))$$

which is equivalent to our original algebra expression but generates smaller intermediate relations. We can carry out this transformation by using rule 7.a. Remember that the rule merely says that the two expressions are equivalent; it does not say that one is better than the other.

Multiple equivalence rules can be used, one after the other, on a query or on parts of the query. As an illustration, suppose that we modify our original query to restrict attention to instructors who have taught a course in 2017. The new relational-algebra query is:

$$\Pi_{name,title}\,(\sigma_{dept_name = \text{“Music”} \wedge year\ =\ 2017}\,(instructor \bowtie (teaches \bowtie \Pi\ \ (course))))$$

Telegram: @uni_k

www.konkur.in

We cannot apply the selection predicate directly to the *instructor* relation, since the predicate involves attributes of both the *instructor* and *teaches* relations. However, we can first apply rule 6.a (associativity of natural join) to transform the join $instructor \bowtie (teaches \bowtie \Pi_{course_id,title}(course))$ into $(instructor \bowtie teaches) \bowtie \Pi_{course_id,title}(course)$:

$$\Pi_{name,title}\ (\sigma_{dept_name = \text{“Music”} \wedge year\ =\ 2017} ((instructor \bowtie teaches) \bowtie \Pi_{course_id,title}(course)))$$

Then, using rule 7.a, we can rewrite our query as:

$$\Pi_{name,title}\ ((\sigma_{dept_name = \text{“Music”} \wedge year\ =\ 2017} (instructor \bowtie teaches)) \bowtie \Pi_{course_id,title}(course))$$

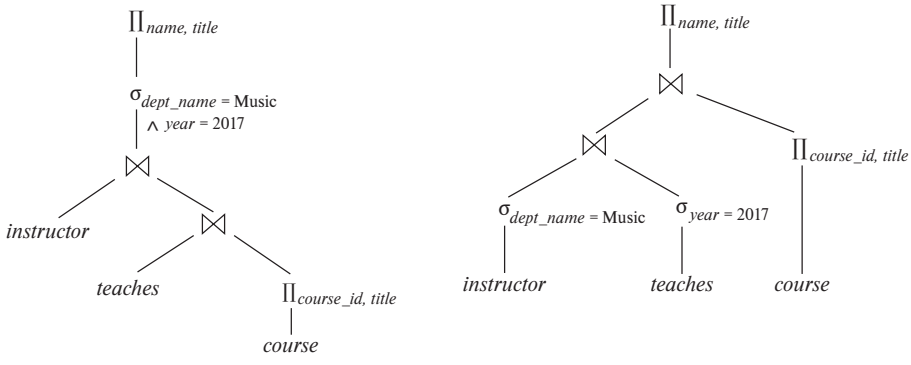
Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections to get the following subexpression:

$$\sigma_{dept_name = \text{“Music”}}\ (\sigma_{year\ =\ 2017}\ (instructor \bowtie teaches))$$

Both of the preceding expressions select tuples with *dept_name* = “Music” and *course_id* = 2017. However, the latter form of the expression provides a new opportunity to apply rule 7.a (“perform selections early”), resulting in the subexpression:

$$\sigma_{dept_name = \text{“Music”}}\ (instructor) \bowtie \sigma_{year\ =\ 2017}\ (teaches)$$

Figure 16.4 depicts the initial expression and the final expression after all these transformations. We could equally well have used rule 7.b to get the final expression directly, without using rule 1 to break the selection into two selections. In fact, rule 7.b can itself be derived from rules 1 and 7.a.



(a) Initial expression tree

(b) Tree after multiple transformations

Figure 16.4 Multiple transformations.

Telegram: @uni_k

www.konkur.in

A set of equivalence rules is said to be **minimal** if no rule can be derived from any combination of the others. The preceding example illustrates that the set of equivalence rules in Section 16.2.1 is not minimal. An expression equivalent to the original expression may be generated in different ways; the number of different ways of generating an expression increases when we use a nonminimal set of equivalence rules. Query optimizers therefore use minimal sets of equivalence rules.

Now consider the following form of our example query:

$$\Pi_{name,title}\ ((\sigma_{dept_name = \text{“Music”}}\ (instructor) \bowtie teaches) \bowtie \Pi_{course_id,title}(course))$$

When we compute the subexpression:

$$(\sigma_{dept_name = \text{“Music”}}\ (instructor) \bowtie teaches)$$

we obtain a relation whose schema is:

$$(ID, name, dept_name, salary, course_id, sec_id, semester, year)$$

We can eliminate several attributes from the schema by pushing projections based on equivalence rules 8.a and 8.b. The only attributes that we must retain are those that either appear in the result of the query or are needed to process subsequent operations. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, we reduce the size of the intermediate result. In our example, the only attributes we need from the join of *instructor* and *teaches* are *name* and *course_id*. Therefore, we can modify the expression to:

$$\Pi_{name,title}\ ((\Pi_{name,course_id}\ ((\sigma_{dept_name = \text{“Music”}}\ (instructor)) \bowtie teaches)) \bowtie \Pi_{course_id,title}(course))$$

The projection $\Pi_{name,course_id}$ reduces the size of the intermediate join results.

16.2.3 Join Ordering

A good ordering of join operations is important for reducing the size of temporary results; hence, most query optimizers pay a lot of attention to the join order. As mentioned in equivalence rule 6.a, the natural-join operation is associative. Thus, for all relations r_1, r_2, and r_3:

$$(r_1 \bowtie r_2) \bowtie r_3 \equiv r_1 \bowtie (r_2 \bowtie r_3)$$

Although these expressions are equivalent, the costs of computing them may differ. Consider again the expression:

$$\Pi_{name,title}\ ((\sigma_{dept_name = \text{“Music”}}\ (instructor)) \bowtie teaches \bowtie \Pi \qquad (course))$$

Telegram: @uni_k

www.konkur.in

We could choose to compute $teaches \bowtie \Pi_{course_id,title}(course)$ first, and then to join the result with:

$$\sigma_{dept_name = \text{“Music”}}(instructor)$$

However, $teaches \bowtie \Pi_{course_id,title}(course)$ is likely to be a large relation, since it contains one tuple for every course taught. In contrast:

$$\sigma_{dept_name = \text{“Music”}}(instructor) \bowtie teaches$$

is probably a small relation. To see that it is, we note that a university has fewer instructors than courses and, since a university has a large number of departments, it is likely that only a small fraction of the university instructors are associated with the Music department. Thus, the preceding expression results in one tuple for each course taught by an instructor in the Music department. Therefore, the temporary relation that we must store is smaller than it would have been had we computed $teaches \bowtie \Pi_{course_id,title}(course)$ first.

There are other options to consider for evaluating our query. We do not care about the order in which attributes appear in a join, since it is easy to change the order before displaying the result. Thus, for all relations r_1 and r_2:

$$r_1 \bowtie r_2 \equiv r_2 \bowtie r_1$$

That is, natural join is commutative (equivalence rule 5).

Using the associativity and commutativity of the natural join (rules 5 and 6), consider the following relational-algebra expression:

$$(instructor \bowtie \Pi_{course_id,title}(course)) \bowtie teaches$$

Note that there are no attributes in common between $\Pi_{course_id,title}(course)$ and *instructor*, so the join is just a Cartesian product. If there are a tuples in *instructor* and b tuples in $\Pi_{course_id,title}(course)$, this Cartesian product generates $a * b$ tuples, one for every possible pair of instructor tuple and course (without regard for whether the instructor taught the course). This Cartesian product would produce a very large temporary relation. However, if the user had entered the preceding expression, we could use the associativity and commutativity of the natural join to transform this expression to the more efficient expression:

$$(instructor \bowtie teaches) \bowtie \Pi_{course_id,title}(course)$$

16.2.4 Enumeration of Equivalent Expressions

Query optimizers can use equivalence rules to systematically generate expressions equivalent to the given query expression. The cost of an expression is computed based

Telegram: @uni_k

www.konkur.in


```
procedure genAllEquivalent(E)
EQ = {E}
repeat
    Match each expression E_i in EQ with each equivalence rule R_j
    if any subexpression e_i of E_i matches one side of R_j
        Create a new expression E' which is identical to E_i, except that
            e_i is transformed to match the other side of R_j
        Add E' to EQ if it is not already present in EQ
until no new expression can be added to EQ
```

Figure 16.5 Procedure to generate all equivalent expressions.

on statistics that are discussed in Section 16.3. Cost-based query optimizers, described in Section 16.4 compute the cost of each alternative and pick the least cost alternative.

Conceptually, enumeration of equivalent expressions can be done as outlined in Figure 16.5. The process proceeds as follows: Given a query expression E, the set of equivalent expressions EQ initially contains only E. Now, each expression in EQ is matched with each equivalence rule. If a subexpression e_j of any expression $E_i \in EQ$ (as a special case, e_j could be E_i itself) matches one side of an equivalence rule, the optimizer generates a copy E_k of E_i, in which e_j is transformed to match the other side of the rule, and adds E_k to EQ. This process continues until no more new expressions can be generated. With a properly chosen set of equivalence rules, the set of equivalent expressions is finite, and the process can be guaranteed to terminate.

For example, given an expression $r \bowtie (s \bowtie t)$, the commutativity rule can match the subexpression $(s \bowtie t)$, and would create a new expression $r \bowtie (t \bowtie s)$. The commutativity rule also matches the join at the root of $r \bowtie (s \bowtie t)$, and creates a new expression $(s \bowtie t) \bowtie r$. Associativity and commutativity rules can continue to be applied to generate new expressions. But eventually applying any equivalence rule will only generate expressions that were already generated earlier, and the process will terminate.

The preceding process is extremely costly both in space and in time, but optimizers can greatly reduce both the space and time cost, using two key ideas.

1. If we generate an expression E' from an expression E_1 by using an equivalence rule on subexpression e_i, then E' and E_1 have identical subexpressions except for e_i and its transformation. Even e_i and its transformed version usually share many identical subexpressions. Expression-representation techniques that allow both expressions to point to shared subexpressions can reduce the space requirement significantly.

Telegram: @uni_k

www.konkur.in

2. It is not always necessary to generate every expression that can be generated with the equivalence rules. If an optimizer takes cost estimates of evaluation into account, it may be able to avoid examining some of the expressions, as we shall see in Section 16.4. We can reduce the time required for optimization by using techniques such as these.

With these and other techniques to reduce the optimization time, equivalence rules can be used to enumerate alternative plans, whose costs can be computed; the lowest-cost plan amongst the alternatives is then chosen. We discuss efficient implementation of cost-based query optimization based on equivalence rules in Section 16.4.2.

Some query optimizers use equivalence rules in a heuristic manner. With such an approach, if the left-hand side of an equivalence rule matches a subtree in a query plan, the subtree is rewritten to match the right-hand side of the rule. This process is repeated till the query plan cannot be further rewritten. Rules must be carefully chosen such that the cost decreases when a rule is applied, and rewriting must eventually terminate. Although this approach can be implemented to execute quite fast, there is no guarantee that it will find the optimal plan.

Yet other query optimizers focus on join order selection, which is often a key factor in query cost. We discuss algorithms for join-order optimization in Section 16.4.1.

16.3 Estimating Statistics of Expression Results

The cost of an operation depends on the size and other statistics of its inputs. Given an expression such as $r \bowtie (s \bowtie t)$ to estimate the cost of joining r with $(s \bowtie t)$, we need to have estimates of statistics such as the size of $s \bowtie t$.

In this section, we first list some statistics about database relations that are stored in database-system catalogs, and then show how to use the stored statistics to estimate statistics on the results of various relational operations.

Given a query expression, we consider it as a tree; we can start from the bottom-level operations, and estimate their statistics, and continue the process on higher-level operations, till we reach the root of the tree. The size estimates that we compute as part of these statistics can be used to compute the cost of algorithms for individual operations in the tree, and these costs can be added up to find the cost of an entire query plan, as we saw in Chapter 15.

One thing that will become clear later in this section is that the estimates are not very accurate, since they are based on assumptions that may not hold exactly. A query-evaluation plan that has the lowest estimated execution cost may therefore not actually have the lowest actual execution cost. However, real-world experience has shown that even if estimates are not precise, the plans with the lowest estimated costs usually have actual execution costs that are either the lowest actual execution costs or are close to the lowest actual execution costs.

Telegram: @uni_k

www.konkur.in

16.3.1 Catalog Information

The database-system catalog stores the following statistical information about database relations:

- n_r, the number of tuples in the relation r.
- b_r, the number of blocks containing tuples of relation r.
- l_r, the size of a tuple of relation r in bytes.
- f_r, the blocking factor of relation r—that is, the number of tuples of relation r that fit into one block.
- $V(A, r)$, the number of distinct values that appear in the relation r for attribute A. This value is the same as the size of $\Pi_A(r)$. If A is a key for relation r, $V(A, r)$ is n_r.

The last statistic, $V(A, r)$, can also be maintained for sets of attributes, if desired, instead of just for individual attributes. Thus, given a set of attributes, $\mathcal{A}$, $V(\mathcal{A}, r)$ is the size of $\Pi_{\mathcal{A}}(r)$.

If we assume that the tuples of relation r are stored together physically in a file, the following equation holds:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Statistics about indices, such as the heights of B^+-tree indices and number of leaf pages in the indices, are also maintained in the catalog.

If we wish to maintain accurate statistics, then every time a relation is modified, we must also update the statistics. This update incurs a substantial amount of overhead. Therefore, most systems do not update the statistics on every modification. Instead, they update the statistics during periods of light system load. As a result, the statistics used for choosing a query-processing strategy may not be completely accurate. However, if not too many updates occur in the intervals between the updates of the statistics, the statistics will be sufficiently accurate to provide a good estimation of the relative costs of the different plans.

The statistical information noted here is simplified. Real-world optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans. For instance, most databases store the distribution of values for each attribute as a **histogram**: in a histogram, the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range. Figure 16.6 shows an example of a histogram for an integer-valued attribute that takes values in the range 1 to 25.

As an example of a histogram, the range of values for an attribute *age* of a relation *person* could be divided into 0–9, 10–19, . . . , 90–99 (assuming a maximum age of 99). With each range we store a count of the number of *person* tuples whose *age* values lie in that range.

Telegram: @uni_k

www.konkur.in

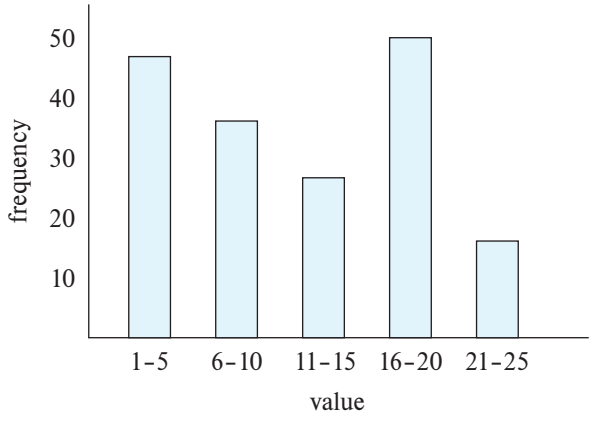


Figure 16.6 Example of histogram.

The histogram shown in Figure 16.6, is an **equi-width histogram** since it divides the range of values into equal-sized ranges. In contrast, an **equi-depth** histogram adjusts the boundaries of the ranges such that each range has the same number of values. Thus, an equi-depth histogram merely stores the boundaries of partitions of the range, and need not store the number of values. For example, the following could be the equidepth histogram for the data whose equi-width histogram is shown in Figure 16.6:

(4, 8, 14, 19)

The histogram indicates that 1/5th of the tuples have age less than 4, another 1/5th have age $\geq$ 4 but $<$ 8, and so on, with the last 1/5th having age $\geq$ 19. Information about the total number of tuples is also stored with the equi-width histogram. Equi-depth histograms are preferred to equi-width histograms since they provide better estimates, and occupy less space.

Histograms used in database systems can also record the number of distinct values in each range, in addition to the number of tuples with attribute values in that range. In our example, the histogram could store the number of distinct age values that lie in each range. Without such histogram information, an optimizer would have to assume that the distribution of values is uniform; that is, each range has the same number of distinct values.

In many database applications, some values are very frequent, compared to other values. To get better estimates for queries that specify these values, many databases store a list of n most frequent values for some n (say 5 or 10), along with the number of times each value appears. In our example, if ages 4, 7, 18, 19, and 23 are the five most frequently occurring values, the database could store the number of persons having each of these ages. The histogram then only stores statistics for age values other than these five values, since we have now have exact counts for these values.

Telegram: @uni_k

www.konkur.in

A histogram takes up only a little space, so histograms on several different attributes can be stored in the system catalog.

16.3.2 Selection Size Estimation

The size estimate of the result of a selection operation depends on the selection predicate. We first consider a single equality predicate, then a single comparison predicate, and finally combinations of predicates.

- $\sigma_{A=a}(r)$: If a is a frequently occurring value for which the occurrence count is available, we can use that value directly as the size estimate for the selection.

  Otherwise if there is no histogram available, we assume uniform distribution of values (i.e., each value appears with equal probability), the selection result is estimated to have $n_r/V(A,r)$ tuples, assuming that the value a appears in attribute A of some record of r. The assumption that the value a in the selection appears in some record is generally true, and cost estimates often make it implicitly. However, it is often not realistic to assume that each value appears with equal probability. The *course_id* attribute in the *takes* relation is an example where the assumption is not valid. It is reasonable to expect that a popular undergraduate course will have many more students than a smaller specialized graduate course. Therefore, certain *course_id* values appear with greater probability than do others. Despite the fact that the uniform-distribution assumption is often not correct, it is a reasonable approximation of reality in many cases, and it helps us to keep our presentation relatively simple.

  If a histogram is available on attribute A, we can locate the range that contains the value a, and modify the above-mentioned estimate $n_r/V(A,r)$ by using the frequency count for that range instead of n_r, and the number of distinct values that occurs in that range instead of $V(A,r)$.

- $\sigma_{A\leq v}(r)$: Consider a selection of the form $\sigma_{A\leq v}(r)$. Suppose that the lowest and highest values ($\min(A,r)$ and $\max(A,r)$) for the attribute are stored in the catalog. Assuming that values are uniformly distributed, we can estimate the number of records that will satisfy the condition $A \leq v$ as:

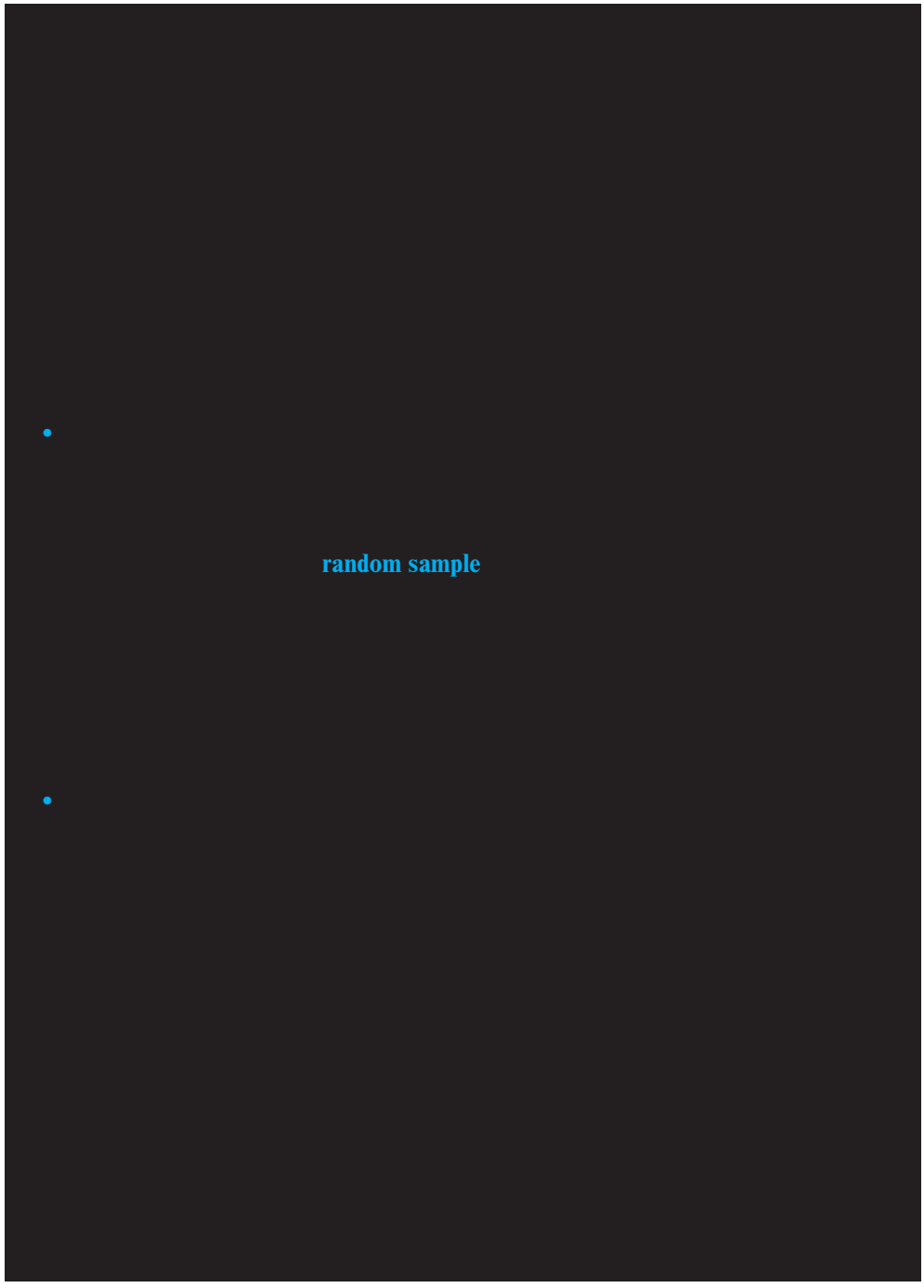
  - 0 if $v < \min(A,r)$
  - n_r if $v \geq \max(A,r)$, and,
  - $n_r \cdot \frac{v-\min(A,r)}{\max(A,r)-\min(A,r)}$, otherwise.

  If a histogram is available on attribute A, we can get a more accurate estimate; we leave the details as an exercise for you.

  In some cases, such as when the query is part of a stored procedure, the value v may not be available when the query is optimized. In such cases, we assume that approximately one-half the records will satisfy the comparison condition. That is,

Telegram: @uni_k

www.konkur.in



Telegram: @uni_k

www.konkur.in

we assume the result has $n_r/2$ tuples; the estimate may be very inaccurate, but it is the best we can do without any further information.

- Complex selections:
  - **Conjunction:** A *conjunctive selection* is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

We can estimate the result size of such a selection: For each θ_i, we estimate the size of the selection $\sigma_{\theta_i}(r)$, denoted by s_i, as described previously. Thus, the probability that a tuple in the relation satisfies selection condition θ_i is s_i/n_r.

The preceding probability is called the **selectivity** of the selection $\sigma_{\theta_i}(r)$. Assuming that the conditions are *independent* of each other, the probability that a tuple satisfies all the conditions is simply the product of all these probabilities. Thus, we estimate the number of tuples in the full selection as:

$$n_r * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

  - **Disjunction:** A *disjunctive selection* is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

As before, let s_i/n_r denote the probability that a tuple satisfies condition θ_i. The probability that the tuple will satisfy the disjunction is then 1 minus the probability that it will satisfy *none* of the conditions:

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdots * (1 - \frac{s_n}{n_r})$$

Multiplying this value by n_r gives us the estimated number of tuples that satisfy the selection.

  - **Negation:** In the absence of nulls, the result of a selection $\sigma_{\neg\theta}(r)$ is simply the tuples of r that are not in $\sigma_\theta(r)$. We already know how to estimate the number of tuples in $\sigma_\theta(r)$. The number of tuples in $\sigma_{\neg\theta}(r)$ is therefore estimated to be n_r minus the estimated number of tuples in $\sigma_\theta(r)$.

We can account for nulls by estimating the number of tuples for which the condition θ would evaluate to *unknown*, and subtracting that number from the above estimate, ignoring nulls. Estimating that number would require extra statistics to be maintained in the catalog.

16.3.3 Join Size Estimation

In this section, we see how to estimate the size of the result of a join.

Telegram: @uni_k

www.konkur.in

The Cartesian product $r \times s$ contains $n_r * n_s$ tuples. Each tuple of $r \times s$ occupies $l_r + l_s$ bytes, from which we can calculate the size of the Cartesian product.

Estimating the size of a natural join is somewhat more complicated than estimating the size of a selection or of a Cartesian product. Let $r(R)$ and $s(S)$ be relations.

- If $R \cap S = \emptyset$—that is, the relations have no attribute in common—then $r \bowtie s$ is the same as $r \times s$, and we can use our estimation technique for Cartesian products.
- If $R \cap S$ is a key for R, then we know that a tuple of s will join with at most one tuple from r. Therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s. The case where $R \cap S$ is a key for S is symmetric to the case just described. If $R \cap S$ forms a foreign key of S, referencing R, the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
- The most difficult case is when $R \cap S$ is a key for neither R nor S. In this case, we assume, as we did for selections, that each value appears with equal probability. Consider a tuple t of r, and assume $R \cap S = \{A\}$. We estimate that tuple t produces

$$\frac{n_s}{V(A, s)}$$

tuples in $r \bowtie s$, since this number is the average number of tuples in s with a given value for the attributes A. Considering all the tuples in r, we estimate that there are

$$\frac{n_r * n_s}{V(A, s)}$$

tuples in $r \bowtie s$. Observe that, if we reverse the roles of r and s in the preceding estimate, we obtain an estimate of

$$\frac{n_r * n_s}{V(A, r)}$$

tuples in $r \bowtie s$. These two estimates differ if $V(A, r) \neq V(A, s)$. If this situation occurs, there are likely to be dangling tuples that do not participate in the join. Thus, the lower of the two estimates is probably the more accurate one.

The preceding estimate of join size may be too high if the $V(A, r)$ values for attribute A in r have few values in common with the $V(A, s)$ values for attribute A in s. However, this situation is unlikely to happen in the real world, since dangling tuples either do not exist or constitute only a small fraction of the tuples, in most real-world relations.

More important, the preceding estimate depends on the assumption that each value appears with equal probability. More sophisticated techniques for size estimation have to be used if this assumption does not hold. For example, if we have histograms on the join attributes of both relations, and both histograms have the same ranges, then we can use the above estimation technique within each range,

Telegram: @uni_k

www.konkur.in

using the number of rows with values in the range instead of n_r or n_s, and the number of distinct values in that range, instead of $V(A, r)$ or $V(A, s)$. We then add up the size estimates obtained for each range to get the overall size estimate. We leave the case where both relations have histograms on the join attribute, but the histograms have different ranges, as an exercise for you.

We can estimate the size of a theta join $r \bowtie_\theta s$ by rewriting the join as $\sigma_\theta(r \times s)$ and using the size estimates for Cartesian products along with the size estimates for selections, which we saw in Section 16.3.2.

To illustrate all these ways of estimating join sizes, consider the expression:

$$student \bowtie takes$$

Assume the following catalog information about the two relations:

- $n_{student} = 5000$.
- $n_{takes} = 10000$.
- $V(ID, takes) = 2500$, which implies that only half the students have taken any course (this is unrealistic, but we use it to show that our size estimates are correct even in this case), and on average, each student who has taken a course has taken four courses.

Note that since *ID* is a primary key of *student*, $V(ID, student) = n_{student} = 5000$.

The attribute *ID* in *takes* is a foreign key on *student*, and null values do not occur in *takes.ID*, since *ID* is part of the primary key of *takes*; thus, the size of $student \bowtie takes$ is exactly n_{takes}, which is 10000.

We now compute the size estimates for $student \bowtie takes$ without using information about foreign keys. Since $V(ID, takes) = 2500$ and $V(ID, student) = 5000$, the two estimates we get are $5000 * 10000/2500 = 20000$ and $5000 * 10000/5000 = 10000$, and we choose the lower one. In this case, the lower of these estimates is the same as that which we computed earlier from information about foreign keys.

16.3.4 Size Estimation for Other Operations

Next we outline how to estimate the sizes of the results of other relational-algebra operations.

- **Projection:** The estimated size (number of records or number of tuples) of a projection of the form $\Pi_A(r)$ is $V(A, r)$, since projection eliminates duplicates.
- **Aggregation:** The size of ${}_G\gamma_A(r)$ is simply $V(G, r)$, since there is one tuple in ${}_G\gamma_A(r)$ for each distinct value of G.
- **Set operations:** If the two inputs to a set operation are selections on the same relation, we can rewrite the set operation as disjunctions, conjunctions, or negations. For example, $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ can be rewritten as σ_{θ_1} (r). Similarly, we can rewrite

Telegram: @uni_k

www.konkur.in

intersections as conjunctions, and we can rewrite set difference by using negation, so long as the two relations participating in the set operations are selections on the same relation. We can then use the estimates for selections involving conjunctions, disjunctions, and negation in Section 16.3.2.

If the inputs are not selections on the same relation, we estimate the sizes this way: The estimated size of $r \cup s$ is the sum of the sizes of r and s. The estimated size of $r \cap s$ is the minimum of the sizes of r and s. The estimated size of $r - s$ is the same size as r. All three estimates may be inaccurate, but provide upper bounds on the sizes.

- **Outer join:** The estimated size of $r \rtimes s$ is the size of $r \bowtie s$ plus the size of r; that of $r \ltimes s$ is symmetric, while that of $r \rtimes\ltimes s$ is the size of $r \bowtie s$ plus the sizes of r and s. All three estimates may be inaccurate, but provide upper bounds on the sizes.

16.3.5 Estimation of Number of Distinct Values

The size estimates discussed earlier depend on statistics such as histograms, or at a minimum, the number of distinct values for an attribute. While these statistics can be precomputed and stored for relations in the database, we need to compute them for intermediate results. Note that estimation of the number of sizes and the number of distinct values of attributes in an intermediate result E_i helps us estimate the sizes and number of distinct values of attributes in the next level intermediate results that use E_i.

For selections, the number of distinct values of an attribute (or set of attributes) A in the result of a selection, $V(A, \sigma_\theta(r))$, can be estimated in these ways:

- If the selection condition θ forces A to take on a specified value (e.g., $A = 3$), $V(A, \sigma_\theta(r)) = 1$.
- If θ forces A to take on one of a specified set of values (e.g., $(A = 1 \vee A = 3 \vee A = 4)$), then $V(A, \sigma_\theta(r))$ is set to the number of specified values.
- If the selection condition θ is of the form $A\ op\ v$, where op is a comparison operator, $V(A, \sigma_\theta(r))$ is estimated to be $V(A, r) * s$, where s is the selectivity of the selection.
- In all other cases of selections, we assume that the distribution of A values is independent of the distribution of the values on which selection conditions are specified, and we use an approximate estimate of $\min(V(A, r), n_{\sigma_\theta(r)})$. A more accurate estimate can be derived for this case using probability theory, but the preceding approximation works fairly well.

For joins, the number of distinct values of an attribute (or set of attributes) A in the result of a join, $V(A, r \bowtie s)$, can be estimated in these ways:

Telegram: @uni_k

www.konkur.in

- If all attributes in A are from r, $V(A, r \bowtie s)$ is estimated as $\min(V(A, r), n_{r \bowtie s})$, and similarly if all attributes in A are from s, $V(A, r \bowtie s)$ is estimated to be $\min(V(A, s), n_{r \bowtie s})$.
- If A contains attributes $A1$ from r and $A2$ from s, then $V(A, r \bowtie s)$ is estimated as:

$$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$$

  Note that some attributes may be in $A1$ as well as in $A2$, and $A1 - A2$ and $A2 - A1$ denote, respectively, attributes in A that are only from r and attributes in A that are only from s. Again, more accurate estimates can be derived by using probability theory, but the above approximations work fairly well.

The estimates of distinct values are straightforward for projections: They are the same in $\Pi_A(r)$ as in r. The same holds for grouping attributes of aggregation. For results of **sum**, **count**, and **average**, we can assume, for simplicity, that all aggregate values are distinct. For **min**(A) and **max**(A), the number of distinct values can be estimated as $\min(V(A, r), V(G, r))$, where G denotes the grouping attributes. We omit details of estimating distinct values for other operations.

16.4 Choice of Evaluation Plans

Generation of expressions is only part of the query-optimization process, since each operation in the expression can be implemented with different algorithms. An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated.

Given an evaluation plan, we can estimate its cost using statistics estimated by the techniques in Section 16.3 coupled with cost estimates for various algorithms and evaluation methods described in Chapter 15.

A **cost-based optimizer** explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost. We have seen how equivalence rules can be used to generate equivalent plans. However, cost-based optimization with arbitrary equivalence rules is fairly complicated. We first cover a simpler version of cost-based optimization, which involves only join-order and join algorithm selection, in Section 16.4.1. Then, in Section 16.4.2, we briefly sketch how a general-purpose optimizer based on equivalence rules can be built, without going into details.

Exploring the space of all possible plans may be too expensive for complex queries. Most optimizers include heuristics to reduce the cost of query optimization, at the potential risk of not finding the optimal plan. We study some such heuristics in Section 16.4.3.

Telegram: @uni_k

www.konkur.in

16.4.1 Cost-Based Join-Order Selection

The most common type of query in SQL consists of a join of a few relations, with join predicates and selections specified in the **where** clause. In this section we consider the problem of choosing the optimal join order for such a query.

For a complex join query, the number of different query plans that are equivalent to the query can be large. As an illustration, consider the expression:

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

where the joins are expressed without any ordering. With $n = 3$, there are 12 different join orderings:

$$\begin{array}{llll} r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\ r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\ r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3 \end{array}$$

In general, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. (We leave the computation of this expression for you to do in Exercise 16.12.) For joins involving small numbers of relations, this number is acceptable; for example, with $n = 5$, the number is 1680. However, as n increases, this number rises quickly. With $n = 7$, the number is 665,280; with $n = 10$, the number is greater than 17.6 billion!

Luckily, it is not necessary to generate all the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form:

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

which represents all join orders where r_1, r_2, and r_3 are joined first (in some order), and the result is joined (in some order) with r_4 and r_5. There are 12 different join orders for computing $r_1 \bowtie r_2 \bowtie r_3$, and 12 orders for computing the join of this result with r_4 and r_5. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations $\{r_1, r_2, r_3\}$, we can use that order for further joins with r_4 and r_5, and we can ignore all costlier join orders of $r_1 \bowtie r_2 \bowtie r_3$. Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.

Using this idea, we can develop a *dynamic-programming* algorithm for finding optimal join orders. Dynamic-programming algorithms store results of computations and reuse them, a procedure that can reduce execution time greatly.

We now consider how to find the optimal join order for a set of n relations $S = \{r_1, r_2, \ldots, r_n\}$, where each relation may have selection conditions, and a set of join conditions between the relations r_i is provided. We assume that relations have unique names.

A recursive procedure implementing the dynamic-programming algorithm appears in Figure 16.7 and is invoked as FindBestPlan(S), where S is the set of relations above. The procedure applies selections on individual relations at the earliest possible point,

Telegram: @uni_k

www.konkur.in


```
procedure FindBestPlan(S)
    if (bestplan[S].cost ≠ ∞) /* bestplan[S] already computed */
        return bestplan[S]
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way of
            accessing S using selection conditions (if any) on S.
    else for each non-empty subset S1 of S such that S1 ≠ S
        P1 = FindBestPlan(S1)
        P2 = FindBestPlan(S − S1)
        for each algorithm A for joining the results of P1 and P2
            // For indexed-nested loops join, the outer relation could be P1 or P2.
            // Similarly for hash-join, the build relation could be P1 or P2.
            // We assume the alternatives are considered as separate algorithms.
            // We assume cost of A does not include cost of reading the inputs.
            if algorithm A is indexed nested loops
                Let P_o and P_i denote the outer and inner inputs of A
                if P_i has a single relation r_i, and r_i has an index on the join
                        attributes
                    plan = "execute P_o.plan; join results of P_o and r_i using A",
                            with any selection condition on P_i performed as
                            part of the join condition
                    cost = P_o.cost + cost of A
                else /* Cannot use indexed nested loops join */
                    cost = ∞
            else
                plan = "execute P1.plan; execute P2.plan;
                            join results of P1 and P2 using A"
                cost = P1.cost + P2.cost + cost of A
            if cost < bestplan[S].cost
                bestplan[S].cost = cost
                bestplan[S].plan = plan
    return bestplan[S]
```

Figure 16.7 Dynamic-programming algorithm for join-order optimization.

that is, when the relations are accessed. It is easiest to understand the procedure assuming that all joins are natural joins, although the procedure works unchanged with any join condition. With arbitrary join conditions, the join of two subexpressions is understood to include all join conditions that relate attributes from the two subexpressions.

Telegram: @uni_k

www.konkur.in

The procedure stores the evaluation plans it computes in an associative array *bestplan*, which is indexed by sets of relations. Each element of the associative array contains two components: the cost of the best plan of S, and the plan itself. The value of $bestplan[S].cost$ is assumed to be initialized to ∞ if $bestplan[S]$ has not yet been computed.

The procedure first checks if the best plan for computing the join of the given set of relations S has been computed already (and stored in the associative array *bestplan*); if so, it returns the already computed plan.

If S contains only one relation, the best way of accessing S (taking selections on S, if any, into account) is recorded in *bestplan*. This may involve using an index to identify tuples, and then fetching the tuples (often referred to as an *index scan*), or scanning the entire relation (often referred to as a *relation scan*).[2] If there is any selection condition on S, other than those ensured by an index scan, a selection operation is added to the plan to ensure all selections on S are satisfied.

Otherwise, if S contains more than one relation, the procedure tries every way of dividing S into two disjoint subsets. For each division, the procedure recursively finds the best plans for each of the two subsets. It then considers all possible algorithms for joining the results of the two subsets. Note that since indexed nested loops join can potentially use either input $P1$ or $P2$ as the inner input, we consider the two alternatives as two different algorithms. The choice of build versus probe input also leads us to consider the two choices for hash join as two different algorithms.

The cost of each alternative is considered, and the least cost option chosen. The join cost considered should not include the cost of reading the inputs, since we assume that the input is pipelined from the preceding operators, which could be a relation/index scan, or a preceding join. Recall that some operators, such as hash join, can be treated as having suboperators with a blocking (materialized) edge between them, but with the input and output edges of the join being pipelined. The join cost formulae that we saw in Chapter 15 can be used with appropriate modifications to ignore the cost of reading the input relations. Note that indexed nested loops join is treated differently from other join techniques: the plan as well as the cost are different in this case, since we do not perform a relation/index scan of the inner input, and the index lookup cost is included in the cost of indexed nested loops join.

The procedure picks the cheapest plan from among all the alternatives for dividing S into two sets and the algorithms for joining the results of the two sets. The cheapest plan and its cost are stored in the array *bestplan* and returned by the procedure. The time complexity of the procedure can be shown to be $O(3^n)$ (see Practice Exercise 16.13).

The order in which tuples are generated by the join of a set of relations is important for finding the best overall join order, since it can affect the cost of further joins. For

[2] If an index contains all the attributes of a relation that are used in a query, it is possible to perform an *index-only scan*, which retrieves the required attribute values from the index,

Telegram: @uni_k

www.konkur.in

instance, if merge join is used, a potentially expensive sort operation is required on the input, unless the input is already sorted on the join attribute.

A particular sort order of the tuples is said to be an **interesting sort order** if it could be useful for a later operation. For instance, generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common to only r_1 and r_2 is not useful. Using merge join for computing $r_1 \bowtie r_2 \bowtie r_3$ may be costlier than using some other join technique, but it may provide an output sorted in an interesting sort order.

Hence, it is not sufficient to find the best join order for each subset of the set of n given relations. Instead, we have to find the best join order for each subset, for each interesting sort order of the join result for that subset. The *bestplan* array can now be indexed by $[S, o]$, where S is a set of relations, and o is an interesting sort order. The FindBestPlan function can then be modified to take interesting sort orders into consideration; we leave details as an exercise for you (see Practice Exercise 16.11).

The number of subsets of n relations is 2^n. The number of interesting sort orders is generally not large. Thus, about 2^n join expressions need to be stored. The dynamic-programming algorithm for finding the best join order can be extended to handle sort orders. Specifically, when considering sort-merge join, the cost of sorting has to be added if an input (which may be a relation, or the result of a join operation) is not sorted on the join attribute, but is not added if it is sorted.

The cost of the extended algorithm depends on the number of interesting orders for each subset of relations; since this number has been found to be small in practice, the cost remains at $O(3^n)$. With $n = 10$, this number is around 59,000, which is much better than the 17.6 billion different join orders. More important, the storage required is much less than before, since we need to store only one join order for each interesting sort order of each of 1024 subsets of $r_1, \ldots, r_{10}$. Although both numbers still increase rapidly with n, commonly occurring joins usually have less than 10 relations and can be handled easily.

The code shown in Figure 16.7 actually considers each possible way of dividing S into two disjoint subsets twice, since each of the two subsets can play the role of $S1$. Considering the division twice does not affect correctness, but wastes time. The code can be optimized as follows: find the alphabetically smallest relation r_i in $S1$, and the alphabetically smallest relation r_j in $S - S1$, and execute the loop only if $r_i < r_j$. Doing so ensures that each division is considered only once.

Further, the code also considers all possible join orders, including those that contain Cartesian products; for example, if two relations r_1 and r_3 do not have any join condition linking the two relations, the code will still consider $S = \{r_1, r_3\}$, which will result in a Cartesian product. It is possible to take join conditions into account, and modify the code to only generate divisions that do not result in Cartesian products. This optimization can save a great deal of time for many queries. See the Further Reading section at the end of the chapter for references providing more details on Cartesian-product-free join order enumeration.

Telegram: @uni_k

www.konkur.in

16.4.2 Cost-Based Optimization with Equivalence Rules

The join-order optimization technique we just saw handles the most common class of queries, which perform an inner join of a set of relations. However, many queries use other features, such as aggregation, outer join, and nested queries, which are not addressed by join-order selection, but can be handled by using equivalence rules.

In this section we outline how to create a general-purpose cost-based optimizer based on equivalence rules. Equivalence rules can help explore alternatives with a wide variety of operations, such as outer joins, aggregations, and set operations, as we have seen earlier. Equivalence rules can be added if required for further operations, such as operators that return the top-K results in sorted order.

In Section 16.2.4, we saw how an optimizer could systematically generate all expressions equivalent to the given query. The procedure for generating equivalent expressions can be modified to generate all possible evaluation plans as follows: A new class of equivalence rules, called **physical equivalence rules**, is added that allows a logical operation, such as a join, to be transformed to a physical operation, such as a hash join, or a nested-loops join. By adding such rules to the original set of equivalence rules, the procedure can generate all possible evaluation plans. The cost estimation techniques we have seen earlier can then be used to choose the optimal (i.e., the least-cost) plan.

However, the procedure shown in Section 16.2.4 is very expensive, even if we do not consider generation of evaluation plans. To make the approach work efficiently requires the following:

1. A space-efficient representation of expressions that avoids making multiple copies of the same subexpressions when equivalence rules are applied.
2. E fficient techniques for detecting duplicate derivations of the same expression.
3. A form of dynamic programming based on **memoization**, which stores the optimal query evaluation plan for a subexpression when it is optimized for the first time; subsequent requests to optimize the same subexpression are handled by returning the already memorized plan.
4. Techniques that avoid generating all possible equivalent plans by keeping track of the cheapest plan generated for any subexpression up to any point of time, and pruning away any plan that is more expensive than the cheapest plan found so far for that subexpression.

The details are more complex than we wish to deal with here. This approach was pioneered by the Volcano research project, and the query optimizer of SQL Server is based on this approach. See the bibliographical notes for references containing further information.

16.4.3 Heuristics in Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query optimization can be reduced by

Telegram: @uni_k

www.konkur.in

evaluation plans for a query can be very large, and finding the optimal plan from this set requires a lot of computational effort. Hence, optimizers use **heuristics** to reduce the cost of optimization.

An example of a heuristic rule is the following rule for transforming relational-algebra queries:

- Perform selection operations as early as possible.

A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation. In the first transformation example in Section 16.2, the selection operation was pushed into a join.

We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost. For an example of where it can result in an increase in cost, consider an expression $\sigma_\theta(r \bowtie s)$, where the condition θ refers to only attributes in s. The selection can certainly be performed before the join. However, if r is extremely small compared to s, and if there is an index on the join attributes of s, but no index on the attributes used by θ, then it is probably a bad idea to perform the selection early. Performing the selection early—that is, directly on s—would require doing a scan of all tuples in s. It is probably cheaper, in this case, to compute the join by using the index and then to reject tuples that fail the selection. (This case is specifically handled by the dynamic programming algorithm for join order optimization.)

The projection operation, like the selection operation, reduces the size of relations. Thus, whenever we need to generate a temporary relation, it is advantageous to apply immediately any projections that are possible. This advantage suggests a companion to the "perform selections early" heuristic:

- Perform projections early.

It is usually better to perform selections earlier than projections, since selections have the potential to reduce the sizes of relations greatly, and selections enable the use of indices to access tuples. An example similar to the one used for the selection heuristic should convince you that this heuristic does not always reduce the cost.

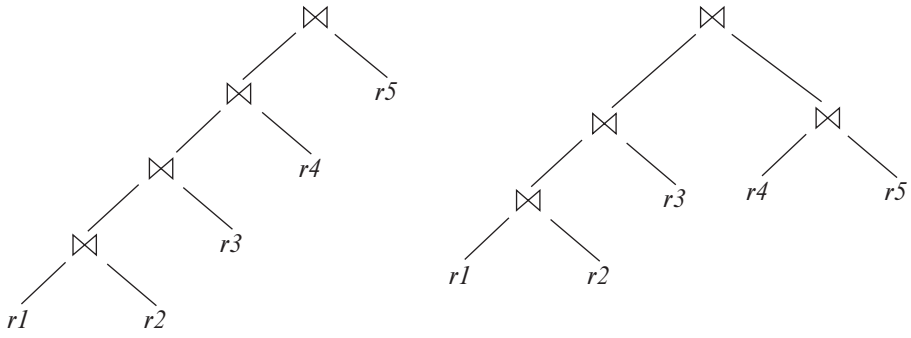
Optimizers based on join-order enumeration typically use heuristic transformations to handle constructs other than joins, and applying the cost-based join-order selection algorithm to subexpressions involving only joins and selections. Details of such heuristics are for the most part specific to individual optimizers, and we do not cover them.

Most practical query optimizers have further heuristics to reduce the cost of optimization. For example, many query optimizers, such as the System R optimizer,[3] do not consider all join orders, but rather restrict the search to particular kinds of join or-

[3]System R was one of the first implementations of SQL, and its optimizer pioneered the idea of cost-based join-order optimization.

Telegram: @uni_k

www.konkur.in



(a) Left-deep join tree (b) Non-left-deep join tree

Figure 16.8 Left-deep join trees.

ders. The System R optimizer considers only those join orders where the right operand of each join is one of the initial relations $r_1, \ldots, r_n$. Such join orders are called **left-deep join orders**. Left-deep join orders are particularly convenient for pipelined evaluation, since the right operand is a stored relation, and thus only one input to each join is pipelined.

Figure 16.8 illustrates the difference between left-deep join trees and non-left-deep join trees. The time it takes to consider all left-deep join orders is $O(n!)$, which is much less than the time to consider all join orders. With the use of dynamic-programming optimizations, the System R optimizer can find the best join order in time $O(n2^n)$. Contrast this cost with the $O(3^n)$ time required to find the best overall join order. The System R optimizer uses heuristics to push selections and projections down the query tree.

A heuristic approach to reduce the cost of join-order selection, which was originally used in some versions of Oracle, works roughly this way: For an n-way join, it considers n evaluation plans. Each plan uses a left-deep join order, starting with a different one of the n relations. The heuristic constructs the join order for each of the n evaluation plans by repeatedly selecting the "best" relation to join next, on the basis of a ranking of the available access paths. Either nested-loop or sort-merge join is chosen for each of the joins, depending on the available access paths. Finally, the heuristic chooses one of the n evaluation plans in a heuristic manner, on the basis of minimizing the number of nested-loop joins that do not have an index available on the inner relation and on the number of sort-merge joins.

Query-optimization approaches that apply heuristic plan choices for some parts of the query, with cost-based choice based on generation of alternative access plans on other parts of the query, have been adopted in several systems. The approach used in System R and in its successor, the Starburst project, is a hierarchical procedure based on the nested-block concept of SQL. The cost-based optimization techniques described here are used for each block of the query separately. The optimizers in several

Telegram: @uni_k

www.konkur.in

database products, such as IBM DB2 and Oracle, are based on the above approach, with extensions to handle other operations such as aggregation. For compound SQL queries (using the ∪, ∩, or − operation), the optimizer processes each component separately and combines the evaluation plans to form the overall evaluation plan.

Most optimizers allow a cost budget to be specified for query optimization. The search for the optimal plan is terminated when the **optimization cost budget** is exceeded, and the best plan found up to that point is returned. The budget itself may be set dynamically; for example, if a cheap plan is found for a query, the budget may be reduced, on the premise that there is no point spending a lot of time optimizing the query if the best plan found so far is already quite cheap. On the other hand, if the best plan found so far is expensive, it makes sense to invest more time in optimization, which could result in a significant reduction in execution time. To best exploit this idea, optimizers usually first apply cheap heuristics to find a plan and then start full cost-based optimization with a budget based on the heuristically chosen plan.

Many applications execute the same query repeatedly, but with different values for the constants. For example, a university application may repeatedly execute a query to find the courses for which a student has registered, but each time for a different student with a different value for the student ID. As a heuristic, many optimizers optimize a query once, with whatever values were provided for the constants when the query was first submitted, and cache the query plan. Whenever the query is executed again, perhaps with new values for constants, the cached query plan is reused (using new values for the constants). The optimal plan for the new constants may differ from the optimal plan for the initial values, but as a heuristic the cached plan is reused.[4] Caching and reuse of query plans is referred to as **plan caching**.

Even with the use of heuristics, cost-based query optimization imposes a substantial overhead on query processing. However, the added cost of cost-based query optimization is usually more than offset by the saving at query-execution time, which is dominated by slow disk accesses. The difference in execution time between a good plan and a bad one may be huge, making query optimization essential. The achieved saving is magnified in those applications that run on a regular basis, where a query can be optimized once, and the selected query plan can be used each time the query is executed. Therefore, most commercial systems include relatively sophisticated optimizers. The bibliographical notes give references to descriptions of the query optimizers of actual database systems.

16.4.4 Optimizing Nested Subqueries

SQL conceptually treats nested subqueries in the **where** clause as functions that take parameters and return either a single value or a set of values (possibly an empty set). The parameters are the variables from an outer-level query that are used in the nested

[4]For the student registration query, the plan would almost certainly be the same for any student ID. But a query that took a range of student IDs, and returned registration information for all student IDs in that range, would probably have a different optimal plan if the range were very small than if the range were

Telegram: @uni_k

www.konkur.in

subquery (these variables are called **correlation variables**). For instance, suppose we have the following query, to find the names of all instructors who taught a course in 2019:

```
select name
from instructor
where exists (select *
              from teaches
              where instructor.ID = teaches.ID
                    and teaches.year = 2019);
```

Conceptually, the subquery can be viewed as a function that takes a parameter (here, *instructor.ID*) and returns the set of all courses taught in 2019 by instructors (with the same *ID*).

SQL evaluates the overall query (conceptually) by computing the Cartesian product of the relations in the outer **from** clause and then testing the predicates in the **where** clause for each tuple in the product. In the preceding example, the predicate tests if the result of the subquery evaluation is empty. In practice, the predicates in the **where** clause that can be used as join predicates, or as selection predicates are evaluated as part of the selections on relations or to perform joins that avoid Cartesian products. Predicates involving nested subqueries in the **where** clause are evaluated subsequently, since they are usually expensive, by invoking the subquery as a function.

The technique of evaluating a nested subquery by invoking it as a function is called **correlated evaluation**. Correlated evaluation is not very efficient, since the subquery is separately evaluated for each tuple in the outer level query. A large number of random disk I/O operations may result.

SQL optimizers therefore attempt to transform nested subqueries into joins, where possible. Efficient join algorithms help avoid expensive random I/O. Where the transformation is not possible, the optimizer keeps the subqueries as separate expressions, optimizes them separately, and then evaluates them by correlated evaluation.

As an attempt at transforming a nested subquery into a join, the query in the preceding example can be rewritten in relational algebra as a join:

$$\Pi_{name}(instructor \bowtie_{instructor.ID=teaches.ID \wedge teaches.year=2019} teaches)$$

Unfortunately, the above query is not quite correct, since the multiset versions of the relational algebra operators are used in SQL implementations, and as a result an instructor who teaches multiple sections in 2019 will appear multiple times in the result of the relational algebra query, although that instructor would appear only once in the SQL query result. Using the set version of the relational algebra operators will not help either, since if there are two instructors with the same name who teach in 2019, the name would appear only once with the set version of relational algebra, but would appear twice in the SQL query result. (We note that the set version of relational algebra

Telegram: @uni_k

www.konkur.in

would give the correct result if the query output contained the primary key of *instructor*, namely *ID*.)

To properly reflect SQL semantics, the number of duplicates of a tuple in the result should not change because of the rewriting. The semijoin operator of the relational algebra provides a solution to this problem. The multiset version of the **semijoin** operator $r \ltimes_\theta s$ is defined as follows: if a tuple r_i appears n times in r, it appears n times in the result of $r \ltimes_\theta$ if there is at least one tuple s_j such that r_i and s_j together satisfy predicate θ; otherwise r_i does not appear in the result. The set version of the semijoin operator $r \ltimes_\theta s$ can be defined as $\Pi_R(r \bowtie_\theta s)$, where R is the set of attributes in the schema of r. The multiset version of the semijoin operator outputs the same tuples, but the number of duplicates of each tuple r_i in the semijoin result is the same as the number of duplicates of r_i in r.

The preceding SQL query can be translated into the following equivalent relational algebra using the multiset semijoin operator:

$$\Pi_{name}(instructor \ltimes_{instructor.ID=teaches.ID \wedge teaches.year=2019} teaches)$$

The above query in the multiset relational algebra gives the same result as the SQL query, including the counts of duplicates. The query can equivalently be written as:

$$\Pi_{name}(instructor \ltimes_{instructor.ID=teaches.ID} (\sigma_{teaches.year=2019}(teaches)))$$

The following SQL query using the **in** clause is equivalent to the preceding SQL query using the **exists** clause, and can be translated to the same relational algebra expression using semijoin.

```
select name
from instructor
where instructor.ID in (select teaches.ID
                        from teaches
                        where teaches.year = 2019);
```

The anti-semijoin is useful with **not exists** queries. The multiset **anti-semijoin** operator $r \overline{\ltimes}_\theta s$ is defined as follows: if a tuple r_i appears n times in r, it appears n times in the result of $r \ltimes_\theta s$ if there does not exist any tuple s_j in s such that r_i and s_j satisfy predicate θ; otherwise r_i does not appear in the result. The anti-semijoin operator is also known as the **anti-join** operator.

Consider the SQL query:

```
select name
from instructor
where not exists (select *
                  from teaches
                  where instructor.ID = teaches.ID
                        and teaches.year = 2019);
```

Telegram: @uni_k

www.konkur.in

The preceding query can be translated into the following relational algebra using the anti-semijoin operator:

$$\Pi_{name}(instructor \overline{\ltimes}_{instructor.ID=teaches.ID}(\sigma_{teaches.year=2019}(teaches)))$$

In general, a query of the form:

select A
from $r_1, r_2, \ldots, r_n$
where P_1 **and exists** (**select** *
from $s_1, s_2, \ldots, s_m$
where P_2^1 **and** P_2^2);

where P_2^1 are predicates that only reference the relations s_i in the subquery, and P_2^2 predicates that also reference the relations r_i from the outer query, can be translated to:

$$\Pi_A((\sigma_{P_1}(r_1 \times r_2 \times \ldots \times r_n)) \ltimes_{P_2^2} \sigma_{P_2^1}(s_1 \times s_2 \times \ldots \times s_m))$$

If **not exists** were used instead of **exists**, the semijoin should be replaced by anti-semijoin in the relational algebra query. If an **in** clause is used instead of **exists**, the relational algebra query can be appropriately modified by adding a corresponding predicate in the semijoin predicate, as our earlier example illustrated.

The process of replacing a nested query by a query with a join, semijoin, or anti-semijoin is called **decorrelation**. The semijoin and anti-semijoin operators can be efficiently implemented using modifications of the join algorithms, as explored in Practice Exercise 15.10.

Consider the following query with aggregation in a scalar subquery, that finds instructors who have taught more than one course section in 2019.

select *name*
from *instructor*
where 1 < (**select count**(*)
from *teaches*
where *instructor.ID* = *teaches.ID*
and *teaches.year* = 2019);

The above query can be rewritten using a semijoin as follows:

$$\Pi_{name}(instructor \ltimes_{(instructor.ID=TID) \wedge (1<cnt)}({}_{ID\ \mathbf{as}\ TID}\gamma_{\mathbf{count}(*)\ \mathbf{as}\ cnt}(\sigma_{year=2019}(teaches))))$$

Observe that the subquery has a predicate *instructor.ID*= *teaches.ID*, and aggregation without a group by clause. The decorrelated query has the predicate moved into the semijoin condition, and the aggregation is now grouped by *ID*. The predicate 1 < (subquery) has turned into a semijoin predicate. Intuitively, the subquery performs a sep-

Telegram: @uni_k

www.konkur.in

arate count for each *instructor.ID*; grouping by *ID* ensures that counts are computed separately for each *ID*.

Decorrelation is clearly more complicated when the nested subquery uses aggregation, or when the nested subquery is used as a scalar subquery. In fact, decorrelation is not possible for certain cases of subqueries. For example, a subquery that is used as a scalar subquery is expected to return only one result; if it returns more than one result, a runtime exception can occur, which is not possible with a decorrelated query. Further, whether to decorrelate or not should ideally be done in a cost-based manner, depending on whether decorrelation reduces the cost or not. Some query optimizers represent nested subqueries using extended relational-algebra constructs, and express transformations of nested subqueries to semijoin, anti-semijoin, and so forth, as equivalence rules. We do not attempt to give algorithms for the general case, and instead refer you to relevant items in the online bibliographical notes.

Optimization of complex nested subqueries is a complicated task, as you can infer from the preceding discussion, and many optimizers do only a limited amount of decorrelation. It better to avoid using complex nested subqueries, where possible, since we cannot be sure that the query optimizer will succeed in converting them to a form that can be evaluated efficiently.

16.5 Materialized Views

When a view is defined, normally the database stores only the query defining the view. In contrast, a **materialized view** is a view whose contents are computed and stored. Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents. However, it is much cheaper in many cases to read the contents of a materialized view than to compute the contents of the view by executing the query defining the view.

Materialized views are important for improving performance in some applications. Consider this view, which gives the total salary in each department:

```
create view department_total_salary(dept_name, total_salary) as
select dept_name, sum (salary)
from instructor
group by dept_name;
```

Suppose the total salary amount at a department is required frequently. Computing the view requires reading every *instructor* tuple pertaining to a department and summing up the salary amounts, which can be time-consuming. In contrast, if the view definition of the total salary amount were materialized, the total salary amount could be found by looking up a single tuple in the materialized view.[5]

[5]The difference may not be all that large for a medium-sized university, very large. For example, if the materialized view computed total sales of

Telegram: @uni_k

www.konkur.in

16.5.1 View Maintenance

A problem with materialized views is that they must be kept up-to-date when the data used in the view definition changes. For instance, if the *salary* value of an instructor is updated, the materialized view will become inconsistent with the underlying data, and it must be updated. The task of keeping a materialized view up-to-date with the underlying data is known as **view maintenance**.

Views can be maintained by manually written code: That is, every piece of code that updates the *salary* value can be modified to also update the total salary amount for the corresponding department. However, this approach is error prone, since it is easy to miss some places where the *salary* is updated, and the materialized view will then no longer match the underlying data.

Another option for maintaining materialized views is to define triggers on insert, delete, and update of each relation in the view definition. The triggers must modify the contents of the materialized view, to take into account the change that caused the trigger to fire. A simplistic way of doing so is to completely recompute the materialized view on every update.

A better option is to modify only the affected parts of the materialized view, which is known as **incremental view maintenance**. We describe how to perform incremental view maintenance in Section 16.5.2.

Modern database systems provide more direct support for incremental view maintenance. Database-system programmers no longer need to define triggers for view maintenance. Instead, once a view is declared to be materialized, the database system computes the contents of the view and incrementally updates the contents when the underlying data change.

Most database systems perform **immediate view maintenance**; that is, incremental view maintenance is performed as soon as an update occurs, as part of the updating transaction. Some database systems also support **deferred view maintenance**, where view maintenance is deferred to a later time; for example, updates may be collected throughout a day, and materialized views may be updated at night. This approach reduces the overhead on update transactions. However, materialized views with deferred view maintenance may not be consistent with the underlying relations on which they are defined.

16.5.2 Incremental View Maintenance

To understand how to maintain materialized views incrementally, we start off by considering individual operations, and then we see how to handle a complete expression.

The changes to a relation that can cause a materialized view to become out-of-date are inserts, deletes, and updates. To simplify our description, we replace updates to a tuple by deletion of the tuple followed by insertion of the updated tuple. Thus, we need

of millions of tuples, the difference between computing the aggregate from the underlying data and looking up the materialized view can be many orders of magnitude.

Telegram: @uni_k

www.konkur.in

to consider only inserts and deletes. The changes (inserts and deletes) to a relation or expression are referred to as its **differential**.

16.5.2.1 Join Operation

Consider the materialized view $v = r \bowtie s$. Suppose we modify r by inserting a set of tuples denoted by i_r. If the old value of r is denoted by r^{old}, and the new value of r by r^{new}, $r^{new} = r^{old} \cup i_r$. Now, the old value of the view, v^{old}, is given by $r^{old} \bowtie s$, and the new value v^{new} is given by $r^{new} \bowtie s$. We can rewrite $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$, which we can again rewrite as $(r^{old} \bowtie s) \cup (i_r \bowtie s)$. In other words:

$$v^{new} = v^{old} \cup (i_r \bowtie s)$$

Thus, to update the materialized view v, we simply need to add the tuples $i_r \bowtie s$ to the old contents of the materialized view. Inserts to s are handled in an exactly symmetric fashion.

Now suppose r is modified by deleting a set of tuples denoted by d_r. Using the same reasoning as above, we get:

$$v^{new} = v^{old} - (d_r \bowtie s)$$

Deletes on s are handled in an exactly symmetric fashion.

16.5.2.2 Selection and Projection Operations

Consider a view $v = \sigma_\theta(r)$. If we modify r by inserting a set of tuples i_r, the new value of v can be computed as:

$$v^{new} = v^{old} \cup \sigma_\theta(i_r)$$

Similarly, if r is modified by deleting a set of tuples d_r, the new value of v can be computed as:

$$v^{new} = v^{old} - \sigma_\theta(d_r)$$

Projection is a more difficult operation with which to deal. Consider a materialized view $v = \Pi_A(r)$. Suppose the relation r is on the schema $R = (A, B)$, and r contains two tuples $(a, 2)$ and $(a, 3)$. Then, $\Pi_A(r)$ has a single tuple (a). If we delete the tuple $(a, 2)$ from r, we cannot delete the tuple (a) from $\Pi_A(r)$: If we did so, the result would be an empty relation, whereas in reality $\Pi_A(r)$ still has a single tuple (a). The reason is that the same tuple (a) is derived in two ways, and deleting one tuple from r removes only one of the ways of deriving (a); the other is still present.

This reason also gives us the intuition for a solution: For each tuple in a projection such as $\Pi_A(r)$, we will keep a count of how many times it was derived.

Telegram: @uni_k

www.konkur.in

When a set of tuples d_r is deleted from r, for each tuple t in d_r we do the following: Let $t.A$ denote the projection of t on the attribute A. We find $(t.A)$ in the materialized view and decrease the count stored with it by 1. If the count becomes 0, $(t.A)$ is deleted from the materialized view.

Handling insertions is relatively straightforward. When a set of tuples i_r is inserted into r, for each tuple t in i_r we do the following: If $(t.A)$ is already present in the materialized view, we increase the count stored with it by 1. If not, we add $(t.A)$ to the materialized view, with the count set to 1.

16.5.2.3 Aggregation Operations

Aggregation operations proceed somewhat like projections. The aggregate operations in SQL are **count, sum, avg, min,** and **max**:

- **count**: Consider a materialized view $v = {}_G\gamma_{count(B)}(r)$, which computes the count of the attribute B, after grouping r by attribute G.

  When a set of tuples i_r is inserted into r, for each tuple t in i_r we do the following: We look for the group $t.G$ in the materialized view. If it is not present, we add $(t.G, 1)$ to the materialized view. If the group $t.G$ is present, we add 1 to the count of the group.

  When a set of tuples d_r is deleted from r, for each tuple t in d_r we do the following: We look for the group $t.G$ in the materialized view and subtract 1 from the count for the group. If the count becomes 0, we delete the tuple for the group $t.G$ from the materialized view.

- **sum**: Consider a materialized view $v = {}_G\gamma_{sum(B)}(r)$.
  When a set of tuples i_r is inserted into r, for each tuple t in i_r we do the following: We look for the group $t.G$ in the materialized view. If it is not present, we add $(t.G, t.B)$ to the materialized view; in addition, we store a count of 1 associated with $(t.G, t.B)$, just as we did for projection. If the group $t.G$ is present, we add the value of $t.B$ to the aggregate value for the group and add 1 to the count of the group.

  When a set of tuples d_r is deleted from r, for each tuple t in d_r we do the following: We look for the group $t.G$ in the materialized view and subtract $t.B$ from the aggregate value for the group. We also subtract 1 from the count for the group, and if the count becomes 0, we delete the tuple for the group $t.G$ from the materialized view.

  Without keeping the extra count value, we would not be able to distinguish a case where the sum for a group is 0 from the case where the last tuple in a group is deleted.

- **avg**: Consider a materialized view $v = {}_G\gamma_{avg(B)}(r)$.
  Directly updating the average on an insert or delete is not possible, since it depends not only on the old average and the tuple being inserted/deleted, but also on the number of tuples in the group.

Telegram: @uni_k

www.konkur.in

Instead, to handle the case of **avg**, we maintain the **sum** and **count** aggregate values as described earlier and compute the average as the sum divided by the count.

- **min, max**: Consider a materialized view $v = {}_G\gamma_{min(B)}(r)$. (The case of **max** is exactly equivalent.)

  Handling insertions on r is straightforward, similar to the case of **sum**. Maintaining the aggregate values **min** and **max** on deletions may be more expensive. For example, if the tuple t corresponding to the minimum value for a group is deleted from r, we have to look at the other tuples of r that are in the same group to find the new minimum value. It is a good idea to create an ordered index on (G, B) since it would help us to find the new minimum value for a group very efficiently.

16.5.2.4 Other Operations

The set operation *intersection* is maintained as follows: Given materialized view $v = r \cap s$, when a tuple is inserted in r we check if it is present in s, and if so we add it to v. If a tuple is deleted from r, we delete it from the intersection if it is present. The other set operations, *union* and *set difference*, are handled in a similar fashion; we leave details to you.

Outer joins are handled in much the same way as joins, but with some extra work. In the case of deletion from r we have to handle tuples in s that no longer match any tuple in r. In the case of insertion to r, we have to handle tuples in s that did not match any tuple in r. Again we leave details to you.

16.5.2.5 Handling Expressions

So far we have seen how to update incrementally the result of a single operation. To handle an entire expression, we can derive expressions for computing the incremental change to the result of each subexpression, starting from the smallest subexpressions.

For example, suppose we wish to incrementally update a materialized view $E_1 \bowtie E_2$ when a set of tuples i_r is inserted into relation r. Let us assume r is used in E_1 alone. Suppose the set of tuples to be inserted into E_1 is given by expression D_1. Then the expression $D_1 \bowtie E_2$ gives the set of tuples to be inserted into $E_1 \bowtie E_2$.

See the online bibliographical notes for further details on incremental view maintenance with expressions.

16.5.3 Query Optimization and Materialized Views

Query optimization can be performed by treating materialized views just like regular relations. However, materialized views offer further opportunities for optimization:

- Rewriting queries to use materialized views:

  Suppose a materialized view $v = r \bowtie s$ is available, and a user submits a query $r \bowtie s \bowtie t$. Rewriting the query as $v \bowtie t$ may provide a more efficient query plan

Telegram: @uni_k

www.konkur.in

than optimizing the query as submitted. Thus, it is the job of the query optimizer to recognize when a materialized view can be used to speed up a query.

- Replacing a use of a materialized view with the view definition:
  Suppose a materialized view $v = r \bowtie s$ is available, but without any index on it, and a user submits a query $\sigma_{A=10}(v)$. Suppose also that s has an index on the common attribute B, and r has an index on attribute A. The best plan for this query may be to replace v with $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$; the selection and join can be performed efficiently by using the indices on $r.A$ and $s.B$, respectively. In contrast, evaluating the selection directly on v may require a full scan of v, which may be more expensive.

The online bibliographical notes give pointers to research showing how to perform query optimization efficiently with materialized views.

16.5.4 Materialized View and Index Selection

Another related optimization problem is that of **materialized view selection**, namely, "What is the best set of views to materialize?" This decision must be made on the basis of the system **workload**, which is a sequence of queries and updates that reflects the typical load on the system. One simple criterion would be to select a set of materialized views that minimizes the overall execution time of the workload of queries and updates, including the time taken to maintain the materialized views. Database administrators usually modify this criterion to take into account the importance of different queries and updates: Fast response may be required for some queries and updates, but a slow response may be acceptable for others.

Indices are just like materialized views, in that they too are derived data, can speed up queries, and may slow down updates. Thus, the problem of **index selection** is closely related to that of materialized view selection, although it is simpler. We examine index and materialized view selection in more detail in Section 25.1.4.1 and Section 25.1.4.2.

Most database systems provide tools to help the database administrator with index and materialized view selection. These tools examine the history of queries and updates and suggest indices and views to be materialized. The Microsoft SQL Server Database Tuning Assistant, the IBM DB2 Design Advisor, and the Oracle SQL Tuning Wizard are examples of such tools.

16.6 Advanced Topics in Query Optimization

There are a number of opportunities for optimizing queries, beyond those we have seen so far. We examine a few of these in this section.

Telegram: @uni_k

www.konkur.in

16.6.1 Top-*K* Optimization

Many queries fetch results sorted on some attributes, and require only the top K results for some K. Sometimes the bound K is specified explicitly. For example, some databases support a **limit** K clause which results in only the top K results being returned by the query. Other databases support alternative ways of specifying similar limits. In other cases, the query may not specify such a limit, but the optimizer may allow a hint to be specified, indicating that only the top K results of the query are likely to be retrieved, even if the query generates more results.

When K is small, a query optimization plan that generates the entire set of results, then sorts and generates the top K, is very inefficient since it discards most of the intermediate results that it computes. Several techniques have been proposed to optimize such *top-K queries*. One approach is to use pipelined plans that can generate the results in sorted order. Another approach is to estimate what is the highest value on the sorted attributes that will appear in the top-K output, and introduce selection predicates that eliminate larger values. If extra tuples beyond the top-K are generated they are discarded, and if too few tuples are generated then the selection condition is changed and the query is re-executed. See the bibliographical notes for references to work on top-K optimization.

16.6.2 Join Minimization

When queries are generated through views, sometimes more relations are joined than are needed for computation of the query. For example, a view v may include the join of *instructor* and *department*, but a use of the view v may use only attributes from *instructor*. The join attribute *dept_name* of *instructor* is a foreign key referencing *department*. Assuming that *instructor.dept_name* has been declared **not null**, the join with *department* can be dropped, with no impact on the query. For under the above assumption, the join with *department* does not eliminate any tuples from *instructor*, nor does it result in extra copies of any *instructor* tuple.

Dropping a relation from a join as above is an example of join minimization. In fact, join minimization can be performed in other situations as well. See the bibliographical notes for references on join minimization.

16.6.3 Optimization of Updates

Update queries often involve subqueries in the **set** and **where** clauses, which must also be taken into account in optimizing the update. Updates that involve a selection on the updated column (e.g., give a 10 percent salary raise to all employees whose salary is $\geq$ \$100,000) must be handled carefully. If the update is done while the selection is being evaluated by an index scan, an updated tuple may be reinserted in the index ahead of the scan and seen again by the scan; the same employee tuple may then get incorrectly updated multiple times (an infinite number of times, in this case). A similar problem also arises with updates involving subqueries whose result is affected by the update.

Telegram: @uni_k

www.konkur.in

The problem of an update affecting the execution of a query associated with the update is known as the **Halloween problem** (named so because it was first recognized on a Halloween day, at IBM). The problem can be avoided by executing the queries defining the update first, creating a list of affected tuples, and updating the tuples and indices as the last step. However, breaking up the execution plan in such a fashion increases the execution cost. Update plans can be optimized by checking if the Halloween problem can occur, and if it cannot occur, updates can be performed while the query is being processed, reducing the update overheads. For example, the Halloween problem cannot occur if the update does not affect index attributes. Even if it does, if the updates decrease the value while the index is scanned in increasing order, updated tuples will not be encountered again during the scan. In such cases, the index can be updated even while the query is being executed, reducing the overall cost.

Update queries that result in a large number of updates can also be optimized by collecting the updates as a batch and then applying the batch of updates separately to each affected index. When applying the batch of updates to an index, the batch is first sorted in the index order for that index; such sorting can greatly reduce the amount of random I/O required for updating indices.

Such optimizations of updates are implemented in most database systems. See the bibliographical notes for references to such optimization.

16.6.4 Multiquery Optimization and Shared Scans

When a batch of queries are submitted together, a query optimizer can potentially exploit common subexpressions between the different queries, evaluating them once and reusing them where required. Complex queries may in fact have subexpressions repeated in different parts of the query, which can be similarly exploited to reduce query evaluation cost. Such optimization is known as **multiquery optimization**.

Common subexpression elimination optimizes subexpressions shared by different expressions in a program by computing and storing the result and reusing it wherever the subexpression occurs. Common subexpression elimination is a standard optimization applied on arithmetic expressions by programming-language compilers. Exploiting common subexpressions among evaluation plans chosen for each of a batch of queries is just as useful in database query evaluation, and is implemented by some databases. However, multiquery optimization can do even better in some cases: A query typically has more than one evaluation plan, and a judiciously chosen set of query evaluation plans for the queries may provide for a greater sharing and lesser cost than that afforded by choosing the lowest cost evaluation plan for each query. More details on multiquery optimization may be found in references cited in the bibliographical notes.

Sharing of relation scans between queries is another limited form of multiquery optimization that is implemented in some databases. The **shared-scan** optimization works as follows: Instead of reading the relation repeatedly from disk, once for each query that needs to scan a relation, data are read once from disk, and pipelined to each of

Telegram: @uni_k

www.konkur.in

the queries. The shared-scan optimization is particularly useful when multiple queries perform a scan on a single large relation (typically a "fact table").

16.6.5 Parametric Query Optimization

Plan caching, which we saw in Section 16.4.3, is used as a heuristic in many databases. Recall that with plan caching, if a query is invoked with some constants, the plan chosen by the optimizer is cached and reused if the query is submitted again, even if the constants in the query are different. For example, suppose a query takes a department name as a parameter and retrieves all courses of the department. With plan caching, a plan chosen when the query is executed for the first time, say for the Music department, is reused if the query is executed for any other department.

Such reuse of plans by plan caching is reasonable if the optimal query plan is not significantly affected by the exact value of the constants in the query. However, if the plan is affected by the value of the constants, parametric query optimization is an alternative.

In **parametric query optimization**, a query is optimized without being provided specific values for its parameters—for example, *dept_name* in the preceding example. The optimizer then outputs several plans, each optimal for a different parameter value. A plan would be output by the optimizer only if it is optimal for some possible value of the parameters. The set of alternative plans output by the optimizer are stored. When a query is submitted with specific values for its parameters, instead of performing a full optimization, the cheapest plan from the set of alternative plans computed earlier is used. Finding the cheapest such plan usually takes much less time than reoptimization. See the bibliographical notes for references on parametric query optimization.

16.6.6 Adaptive Query Processing

As we noted earlier, query optimization is based on estimates that are at best approximations. Thus, it is possible at times for the optimizer to choose a plan that turns out to perform very badly. Adaptive operators that choose the specific operator at execution time provide a partial solution to this problem. For example, SQL Server supports an adaptive join algorithm that checks the size of its outer input, and chooses either nested loops join, or hash join depending on the size of the outer input.

Many systems also include the ability to monitor the behavior of a plan during query execution, and adapt the plan accordingly. For example, suppose the statistics collected by the system during early stages of the plan's execution (or the execution of subparts of the plan) are found to differ substantially from the optimizers estimates to such an extent that it is clear that the chosen plan is suboptimal. Then an adaptive system may abort the execution, choose a new query execution plan using the statistics collected during the initial execution, and restart execution using the new plan; the statistics collected during the execution of the old plan ensure the old plan is not selected again. Further, the system must avoid repeated aborts and restarts; ideally, the system should ensure that the overall cost of query evaluation is close to that with the

Telegram: @uni_k

www.konkur.in

plan that would be chosen if the optimizer had exact statistics. The specific criteria and mechanisms for such adaptive query processing are complex, and are referenced in the bibliographic notes available online.

16.7 Summary

- Given a query, there are generally a variety of methods for computing the answer. It is the responsibility of the system to transform the query as entered by the user into an equivalent query that can be computed more efficiently. The process of finding a good strategy for processing a query is called *query optimization*.
- The evaluation of complex queries involves many accesses to disk. Since the transfer of data from disk is slow relative to the speed of main memory and the CPU of the computer system, it is worthwhile to allocate a considerable amount of processing to choose a method that minimizes disk accesses.
- There are a number of equivalence rules that we can use to transform an expression into an equivalent one. We use these rules to generate systematically all expressions equivalent to the given query.
- Each relational-algebra expression represents a particular sequence of operations. The first step in selecting a query-processing strategy is to find a relational-algebra expression that is equivalent to the given expression and is estimated to cost less to execute.
- The strategy that the database system chooses for evaluating an operation depends on the size of each relation and on the distribution of values within columns. So that they can base the strategy choice on reliable information, database systems may store statistics for each relation r. These statistics include:
  - The number of tuples in the relation r.
  - The size of a record (tuple) of relation r in bytes.
  - The number of distinct values that appear in the relation r for a particular attribute.
- Most database systems use histograms to store the number of values for an attribute within each of several ranges of values. Histograms are often computed using sampling.
- These statistics allow us to estimate the sizes of the results of various operations, as well as the cost of executing the operations. Statistical information about relations is particularly useful when several indices are available to assist in the processing of a query. The presence of these structures has a significant influence on the choice of a query-processing strategy.

Telegram: @uni_k

www.konkur.in

- Alternative evaluation plans for each expression can be generated by equivalence rules, and the cheapest plan across all expressions can be chosen. Several optimization techniques are available to reduce the number of alternative expressions and plans that need to be generated.
- We use heuristics to reduce the number of plans considered, and thereby to reduce the cost of optimization. Heuristic rules for transforming relational-algebra queries include "Perform selection operations as early as possible," "Perform projections early," and "Avoid Cartesian products."
- Materialized views can be used to speed up query processing. Incremental view maintenance is needed to efficiently update materialized views when the underlying relations are modified. The differential of an operation can be computed by means of algebraic expressions involving differentials of the inputs of the operation. Other issues related to materialized views include how to optimize queries by making use of available materialized views, and how to select views to be materialized.
- A number of advanced optimization techniques have been proposed, such as top-K optimization, join minimization, optimization of updates, multiquery optimization, and parametric query optimization.

Review Terms

- Query optimization
- Transformation of expressions
- Equivalence of expressions
- Equivalence rules
  - Join commutativity
  - Join associativity
- Minimal set of equivalence rules
- Enumeration of equivalent expressions
- Statistics estimation
- Catalog information
- Size estimation
  - Selection
  - Selectivity
  - Join
- Histograms
- Distinct value estimation
- Random sample
- Choice of evaluation plans
- Interaction of evaluation techniques
- Cost-based optimization
- Join-order optimization
  - Dynamic-programming algorithm
  - Left-deep join order
  - Interesting sort order
- Heuristic optimization
- Plan caching
- Access-plan selection

Telegram: @uni_k

www.konkur.in

- Correlated evaluation
- Decorrelation
- Semijoin
- Anti-semijoin
- Materialized views
- Materialized view maintenance
  - Recomputation
  - Incremental maintenance
  - Insertion
  - Deletion
  - Updates
- Query optimization with materialized views
- Index selection
- Materialized view selection
- Top-K optimization
- Join minimization
- Halloween problem
- Multiquery optimization

Practice Exercises

16.1 Download the university database schema and the large university dataset from dbbook.com. Create the university schema on your favorite database, and load the large university dataset. Use the **explain** feature described in Note 16.1 on page 746 to view the plan chosen by the database, in different cases as detailed below.

a. Write a query with an equality condition on *student.name* (which does not have an index), and view the plan chosen.

b. Create an index on the attribute *student.name*, and view the plan chosen for the above query.

c. Create simple queries joining two relations, or three relations, and view the plans chosen.

d. Create a query that computes an aggregate with grouping, and view the plan chosen.

e. Create an SQL query whose chosen plan uses a semijoin operation.

f. Create an SQL query that uses a **not in** clause, with a subquery using aggregation. Observe what plan is chosen.

g. Create a query for which the chosen plan uses correlated evaluation (the way correlated evaluation is represented varies by database, but most databases would show a filter or a project operator with a subplan or subquery).

h. Create an SQL update query that updates a single row in a relation. View the plan chosen for the update query.

Telegram: @uni_k

www.konkur.in

i. Create an SQL update query that updates a large number of rows in a relation, using a subquery to compute the new value. View the plan chosen for the update query.

16.2 Show that the following equivalences hold. Explain how you can apply them to improve the efficiency of certain queries:

a. $E_1 \bowtie_\theta (E_2 - E_3) \equiv (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

b. $\sigma_\theta({}_A\gamma_F(E)) \equiv {}_A\gamma_F(\sigma_\theta(E))$, where θ uses only attributes from A.

c. $\sigma_\theta(E_1 \rtimes E_2) \equiv \sigma_\theta(E_1) \rtimes E_2$, where θ uses only attributes from E_1.

16.3 For each of the following pairs of expressions, give instances of relations that show the expressions are not equivalent.

a. $\Pi_A(r - s)$ and $\Pi_A(r) - \Pi_A(s)$.

b. $\sigma_{B<4}({}_A\gamma_{max(B)\ \textbf{as}\ B}(r))$ and ${}_A\gamma_{max(B)\ \textbf{as}\ B}(\sigma_{B<4}(r))$.

c. In the preceding expressions, if both occurrences of *max* were replaced by *min*, would the expressions be equivalent?

d. $(r \bowtie s) \bowtie t$ and $r \bowtie (s \bowtie t)$
In other words, the natural right outer join is not associative.

e. $\sigma_\theta(E_1 \rtimes E_2)$ and $E_1 \rtimes \sigma_\theta(E_2)$, where θ uses only attributes from E_2.

16.4 SQL allows relations with duplicates (Chapter 3), and the multiset version of the relational algebra is defined in Note 3.1 on page 80, Note 3.2 on page 97, and Note 3.3 on page 108. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational algebra.

16.5 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$, with primary keys A, C, and E, respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

16.6 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Practice Exercise 16.5. Assume that there are no primary keys, except the entire schema. Let $V(C, r_1)$ be 900, $V(C, r_2)$ be 1100, $V(E, r_2)$ be 50, and $V(E, r_3)$ be 100. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$ and give an efficient strategy for computing the join.

16.7 Suppose that a B^+-tree index on *building* is available on relation *department* and that no other index is available. What would be the best way to handle the following selections that involve negation?

a. $\sigma_{\neg(building < \text{“Watson”})}(department)$

Telegram: @uni_k

www.konkur.in

b. $\sigma_{\neg (building = \text{“Watson”})}(department)$

c. $\sigma_{\neg (building < \text{“Watson”} \vee budget < 50000)}(department)$

16.8 Consider the query:

```
select *
from r, s
where upper(r.A) = upper(s.A);
```

where “upper” is a function that returns its input argument with all lowercase letters replaced by the corresponding uppercase letters.

a. Find out what plan is generated for this query on the database system you use.

b. Some database systems would use a (block) nested-loop join for this query, which can be very inefficient. Briefly explain how hash-join or merge-join can be used for this query.

16.9 Give conditions under which the following expressions are equivalent:

$$_{A,B}\gamma_{agg(C)}(E_1 \bowtie E_2) \quad \text{and} \quad (_{A}\gamma_{agg(C)}(E_1)) \bowtie E_2$$

where *agg* denotes any aggregation operation. How can the above conditions be relaxed if *agg* is one of **min** or **max**?

16.10 Consider the issue of interesting orders in optimization. Suppose you are given a query that computes the natural join of a set of relations S. Given a subset $S1$ of S, what are the interesting orders of $S1$?

16.11 Modify the FindBestPlan(S) function to create a function FindBestPlan(S, O), where O is a desired sort order for S, and which considers interesting sort orders. A *null* order indicates that the order is not relevant. *Hints*: An algorithm A may give the desired order O; if not a sort operation may need to be added to get the desired order. If A is a merge-join, FindBestPlan must be invoked on the two inputs with the desired orders for the inputs.

16.12 Show that, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. *Hint:* A **complete binary tree** is one where every internal node has exactly two children. Use the fact that the number of different complete binary trees with n leaf nodes is:

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

If you wish, you can derive the formula for the number of complete binary trees with n nodes from the formula for the number of binary trees with n nodes. The number of binary trees with n nodes is:

$$\frac{1}{n+1}\binom{2n}{n}$$

Telegram: @uni_k

www.konkur.in

This number is known as the **Catalan number**, and its derivation can be found in any standard textbook on data structures or algorithms.

16.13 Show that the lowest-cost join order can be computed in time $O(3^n)$. Assume that you can store and look up information about a set of relations (such as the optimal join order for the set, and the cost of that join order) in constant time. (If you find this exercise difficult, at least show the looser time bound of $O(2^{2n})$.)

16.14 Show that, if only left-deep join trees are considered, as in the System R optimizer, the time taken to find the most efficient join order is around $n2^n$. Assume that there is only one interesting sort order.

16.15 Consider the bank database of Figure 16.9, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Write a nested query on the relation *account* to find, for each branch with name starting with B, all accounts with the maximum balance at the branch.

b. Rewrite the preceding query without using a nested subquery; in other words, decorrelate the query, but in SQL.

c. Give a relational algebra expression using semijoin equivalent to the query.

d. Give a procedure (similar to that described in Section 16.4.4) for decorrelating such queries.

Exercises

16.16 Suppose that a B$^+$-tree index on (*dept_name*, *building*) is available on relation *department*. What would be the best way to handle the following selection?

$$\sigma_{(building < \text{“Watson”}) \wedge (budget < 55000) \wedge (dept_name = \text{“Music”})}(department)$$

---

branch(*branch_name*, *branch_city, assets*)
customer (*customer_name*, *customer_street, customer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name, balance*)
depositor (*customer_name*, *account_number*)

Figure 16.9 Banking

Telegram: @uni_k

www.konkur.in

16.17 Show how to derive the following equivalences by a sequence of transformations using the equivalence rules in Section 16.2.1.

a. $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2

16.18 Consider the two expressions $\sigma_\theta(E_1 \rtimes E_2)$ and $\sigma_\theta(E_1 \bowtie E_2)$.

a. Show using an example that the two expressions are not equivalent in general.

b. Give a simple condition on the predicate θ, which if satisfied will ensure that the two expressions are equivalent.

16.19 A set of equivalence rules is said to be *complete* if, whenever two expressions are equivalent, one can be derived from the other by a sequence of uses of the equivalence rules. Is the set of equivalence rules that we considered in Section 16.2.1 complete? Hint: Consider the equivalence $\sigma_{3=5}(r) \equiv \{\ \}$.

16.20 Explain how to use a histogram to estimate the size of a selection of the form $\sigma_{A \leq v}(r)$.

16.21 Suppose two relations r and s have histograms on attributes $r.A$ and $s.A$, respectively, but with different ranges. Suggest how to use the histograms to estimate the size of $r \bowtie s$. Hint: Split the ranges of each histogram further.

16.22 Consider the query

```
select A, B
from   r
where r.B < some (select B
                  from   s
                  where s.A = r.A)
```

Show how to decorrelate this query using the multiset version of the semi join operation.

16.23 Describe how to incrementally maintain the results of the following operations on both insertions and deletions:

a. Union and set difference.

b. Left outer join.

16.24 Give an example of an expression defining a materialized view and two situations (sets of statistics for the input relations and the differentials) such that incremental view maintenance is better than recomputation in one situation, and recomputation is better in the other situation.

Telegram: @uni_k

www.konkur.in

16.25 Suppose you want to get answers to $r \bowtie s$ sorted on an attribute of r, and want only the top K answers for some relatively small K. Give a good way of evaluating the query:

a. When the join is on a foreign key of r referencing s, where the foreign key attribute is declared to be not null.

b. When the join is not on a foreign key.

16.26 Consider a relation $r(A, B, C)$, with an index on attribute A. Give an example of a query that can be answered by using the index only, without looking at the tuples in the relation. (Query plans that use only the index, without accessing the actual relation, are called *index-only* plans.)

16.27 Suppose you have an update query U. Give a simple sufficient condition on U that will ensure that the Halloween problem cannot occur, regardless of the execution plan chosen or the indices that exist.

Further Reading

The seminal work of [Selinger et al. (1979)] describes access-path selection in the System R optimizer, which was one of the earliest relational-query optimizers. Query processing in Starburst, described in [Haas et al. (1989)], forms the basis for query optimization in IBM DB2.

[Graefe and McKenna (1993)] describes Volcano, an equivalence-rule–based query optimizer that, along with its successor Cascades ([Graefe (1995)]), forms the basis of query optimization in Microsoft SQL Server. [Moerkotte (2014)] provides extensive textbook coverage of query optimization, including optimizations of the dynamic programming algorithm for join order optimization to avoid considering Cartesian products. Avoiding generation of plans with Cartesian products can result in substantial reduction in optimization cost for common queries.

The bibliographic notes for this chapter, available online, provides references to research on a variety of optimization techniques, including optimization of queries with aggregates, with outer joins, nested subqueries, top-K queries, join minimization, optimization of update queries, materialized view maintenance and view matching, index and materialized view selection, parametric query optimization, and multiquery optimization.

Bibliography

[Graefe (1995)] G. Graefe, "The Cascades Framework for Query Optimization", *Data Engineering Bulletin*, Volume 18, Number 3 (1995), pages 19–29.

Telegram: @uni_k

www.konkur.in

[Graefe and McKenna (1993)] G. Graefe and W. McKenna, "The Volcano Optimizer Generator", In *Proc. of the International Conf. on Data Engineering* (1993), pages 209–218.

[Haas et al. (1989)] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989), pages 377–388.

[Moerkotte (2014)] G. Moerkotte, *Building Query Compilers*, available online at http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, retrieved 13 Dec 2018 (2014).

[Selinger et al. (1979)] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database System", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), pages 23–34.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



PART 7

TRANSACTION MANAGEMENT

The term *transaction* refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

It is important that either all actions of a transaction be executed completely, or, in case of some failure, partial effects of each incomplete transaction be undone. This property is called *atomicity*. Further, once a transaction is successfully executed, its effects must persist in the database—a system failure should not result in the database forgetting about a transaction that successfully completed. This property is called *durability*.

In a database system where multiple transactions are executing concurrently, if updates to shared data are not controlled, there is potential for transactions to see inconsistent intermediate states created by updates of other transactions. Such a situation can result in erroneous updates to data stored in the database. Thus, database systems must provide mechanisms to isolate transactions from the effects of other concurrently executing transactions. This property is called *isolation*.

Chapter 17 describes the concept of a transaction in detail, including the properties of atomicity, durability, isolation, and other properties provided by the transaction abstraction. In particular, the chapter makes precise the notion of isolation by means of a concept called serializability.

Chapter 18 describes several concurrency-control techniques that help implement the isolation property. Chapter 19 describes the recovery management component of a database, which implements the atomicity and durability properties.

Taken as a whole, the transaction-management component of a database system allows application developers to focus on the implementation of individual transactions, ignoring the issues of concurrency and fault tolerance.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 17



Transactions

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations. It is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited but the savings account not credited.

Collections of operations that form a single logical unit of work are called transactions. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total balance might see the checking-account balance before it is debited by the funds-transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

This chapter introduces the basic concepts of transaction processing. Details on concurrent transaction processing and recovery from failures are in Chapter 18 and Chapter 19, respectively.

17.1 Transaction Concept

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (e.g., C++ or Java), with embedded database accesses in JDBC or ODBC. A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

This collection of steps must appear to the user as a single, indivisible unit. Since a transaction is indivisible, it either executes in its entirety or not at all. Thus, if a

Telegram: @uni_k

www.konkur.in

transaction begins to execute but fails for whatever reason, any changes to the database that the transaction may have made must be undone. This requirement holds regardless of whether the transaction itself failed (e.g., if it divided by zero), the operating system crashed, or the computer itself stopped operating. As we shall see, ensuring that this requirement is met is difficult since some changes to the database may still be stored only in the main-memory variables of the transaction, while others may have been written to the database and stored on disk. This "all-or-none" property is referred to as **atomicity**.

Furthermore, since a transaction is a single unit, its actions cannot appear to be separated by other database operations not part of the transaction. While we wish to present this user-level impression of transactions, we know that reality is quite different. Even a single SQL statement involves many separate accesses to the database, and a transaction may consist of several SQL statements. Therefore, the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as **isolation**.

Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system "forgets" about the transaction. Thus, a transaction's actions must persist across crashes. This property is referred to as **durability**.

Because of the above three properties, transactions are an ideal way of structuring interaction with a database. This leads us to impose a requirement on transactions themselves. A transaction must preserve database consistency—if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction. This consistency requirement goes beyond the data-integrity constraints we have seen earlier (such as primary-key constraints, referential integrity, **check** constraints, and the like). Rather, transactions are expected to go beyond that to ensure preservation of those application-dependent consistency constraints that are too complex to state using the SQL constructs for data integrity. How this is done is the responsibility of the programmer who codes a transaction. This property is referred to as **consistency**.

To restate the above more concisely, we require that the database system maintain the following properties of the transactions:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**. Execution of a transaction in isolation (i.e., with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.

Telegram: @uni_k

www.konkur.in

Thus, each transaction is unaware of other transactions executing concurrently in the system.

- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

As we shall see later, ensuring the isolation property may have a significant adverse effect on system performance. For this reason, some applications compromise on the isolation property. We shall study these compromises after first studying the strict enforcement of the ACID properties.

17.2 A Simple Transaction Model

Because SQL is a powerful and complex language, we begin our study of transactions with a simple database language that focuses on when data are moved from disk to main memory and from main memory to disk. In doing this, we ignore SQL **insert** and **delete** operations and defer considering them until Section 18.4. The only actual operations on the data are restricted in our simple language to arithmetic operations. Later we shall discuss transactions in a realistic, SQL-based context with a richer set of operations. The data items in our simplified model contain a single data value (a number in our examples). Each data item is identified by a name (typically a single letter in our examples, that is, A, B, C, etc.).

We shall illustrate the transaction concept using a simple bank application consisting of several accounts and a set of transactions that access and update those accounts. Transactions access data using two operations:

- read(X), which transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.
- write(X), which transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

It is important to know if a change to a data item appears only in main memory or if it has been written to the database on disk. In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored elsewhere and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately. We discuss storage issues further in Section 17.3 and discuss the issue of when database data in main memory are written to the database on disk in Chapter 19.

Telegram: @uni_k

www.konkur.in

Let T_i be a transaction that transfers \$50 from account A to account B. This transaction can be defined as:

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let us now consider each of the ACID properties. (For ease of presentation, we consider them in an order different from the order A-C-I-D.)

- **Consistency**: The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

  Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints, as we discussed in Section 4.4.

- **Atomicity**: Suppose that, just before the execution of transaction T_i, the values of accounts A and B are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction T_i, a failure occurs that prevents T_i from completing its execution successfully. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

  Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction T_i is executed to completion, there exists a point at which the value of account A is \$950 and the value of account B is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is \$950, and the value of account B is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

Telegram: @uni_k

www.konkur.in

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*. If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed. We discuss these ideas further in Section 17.4. Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system**, which we describe in detail in Chapter 19.

- **Durability**: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

  We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. Protection against loss of data on disk is discussed in Chapter 19. We can guarantee durability by ensuring that either:

  1. The updates carried out by the transaction have been written to disk before the transaction completes.
  2. Information about the updates carried out by the transaction is written to disk, and such information is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

  The recovery system of the database, described in Chapter 19, is responsible for ensuring durability, in addition to ensuring atomicity.

- **Isolation**: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

  For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes $A + B$, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

  A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as we shall see in Section

Telegram: @uni_k

www.konkur.in

17.5. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

We discuss the problems caused by concurrently executing transactions in Section 17.5. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. We shall discuss the principles of isolation further in Section 17.6. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system**, which we discuss in Chapter 18.

17.3 Storage Structure

To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.

In Chapter 12, we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or non-volatile storage. We review these terms and introduce another class of storage, called stable storage.

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access any data item in volatile storage directly.
- **Non-volatile storage**. Information residing in non-volatile storage survives system crashes. Examples of non-volatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media and magnetic tapes, used for archival storage. At the current state of technology, non-volatile storage is slower than volatile storage, particularly for random access. Both secondary and tertiary storage devices, however, are susceptible to failures that may result in loss of information.
- **Stable storage**. Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically *never* cannot be guaranteed—for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. To implement stable storage, we replicate the information in several non-volatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information. Section 19.2.1 discusses stable-storage implementation.

Telegram: @uni_k

www.konkur.in

The distinctions among the various storage types can be less clear in practice than in our presentation. For example, certain systems, for example some RAID controllers, provide battery backup, so that some main memory can survive system crashes and power failures.

For a transaction to be durable, its changes need to be written to stable storage. Similarly, for a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk. The degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is. In some cases, a single copy on disk is considered sufficient, but applications whose data are highly valuable and whose transactions are highly important require multiple copies, or, in other words, a closer approximation of the idealized concept of stable storage.

17.4 Transaction Atomicity and Durability

As we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**. Each database modification made by a transaction is first recorded in the log. We record the identifier of the transaction performing the modification, the identifier of the data item being modified, and both the old value (prior to modification) and the new value (after modification) of the data item. Only then is the database itself modified. Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution. Details of log-based recovery are discussed in Chapter 19.

A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user and is not handled by the database system.

We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

Telegram: @uni_k

www.konkur.in

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

The state diagram corresponding to a transaction appears in Figure 17.1. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if it has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

As mentioned earlier, we assume for now that failures do not result in loss of data on disk. Chapter 19 discusses techniques to deal with loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (e.g., because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

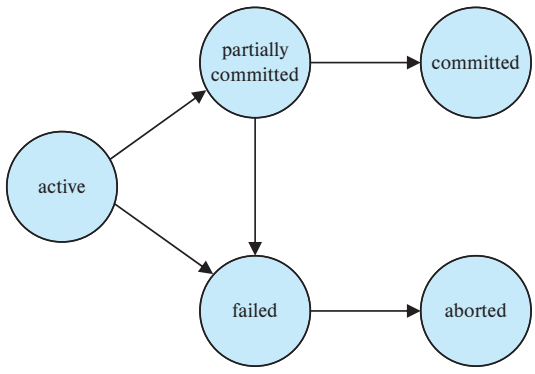


Figure 17.1 State diagram of a transaction.

Telegram: @uni_k

www.konkur.in

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

We must be cautious when dealing with **observable external writes**, such as writes to a user's screen, or sending email. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system. Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in a special relation in the database, and to perform the actual writes only after the transaction enters the committed state. If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in non-volatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example, suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back into the user's account, needs to be executed when the system is restarted.

As another example, consider a user making a booking over the web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits. In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the web application again, she will be able to see whether her transaction had succeeded or not.

For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity.

17.5 Transaction Isolation

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of

Telegram: @uni_k

www.konkur.in



concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.
- **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to un-

Telegram: @uni_k

www.konkur.in

predictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

When several transactions run concurrently, the isolation property may be violated, resulting in database consistency being destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure the isolation property and thus database consistency.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**. We study concurrency-control schemes in Chapter 18; for now, we focus on the concept of correct concurrent execution.

Consider again the simplified banking system of Section 17.1, which has several accounts, and a set of transactions that access and update those accounts. Let T_1 and T_2 be two transactions that transfer funds from one account to another. Transaction T_1 transfers \$50 from account A to account B. It is defined as:

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction T_2 transfers 10 percent of the balance from account A to account B. It is defined as:

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Telegram: @uni_k

www.konkur.in

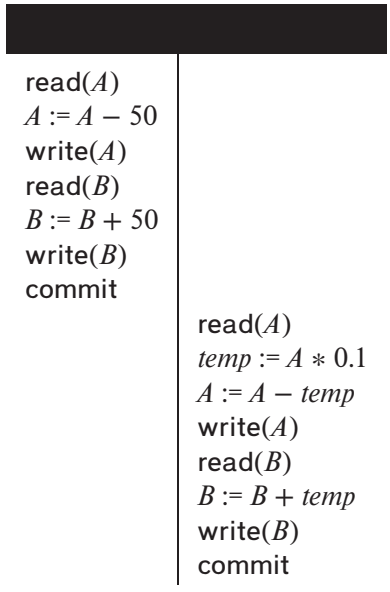


Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

Suppose the current values of accounts A and B are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order T_1 followed by T_2. This execution sequence appears in Figure 17.2. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_1 appearing in the left column and instructions of T_2 appearing in the right column. The final values of accounts A and B, after the execution in Figure 17.2 takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts A and B—that is, the sum $A + B$—is preserved after the execution of both transactions.

Similarly, if the transactions are executed one at a time in the order T_2 followed by T_1, then the corresponding execution sequence is that of Figure 17.3. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions and they must preserve the order in which the instructions appear in each individual transaction. For example, in transaction T_1, the instruction write(A) must appear before the instruction read(B), in any valid schedule. Note that we include in our schedules the **commit** operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence (T_1 followed by T_2) as schedule 1, and to the second execution sequence (T_2 followed by T_1) as schedule 2.

Telegram: @uni_k

www.konkur.in

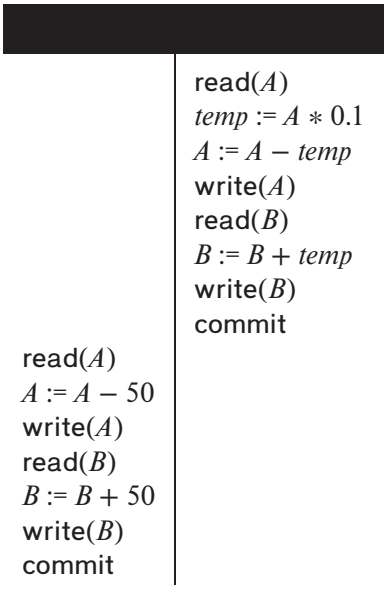


Figure 17.3 Schedule 2—a serial schedule in which T_2 is followed by T_1.

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Recalling a well-known formula from combinatorics, we note that, for a set of n transactions, there exist n factorial ($n!$) different valid serial schedules.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.[1]

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 17.4. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order T_1 followed by T_2. The sum $A + B$ is indeed preserved.

[1]The number of possible schedules for a set of n transactions is very large. There are $n!$ different serial schedules. Considering all the possible ways that steps of transactions might be interleaved, the total number of possible schedules is much larger than $n!$.

Telegram: @uni_k

www.konkur.in

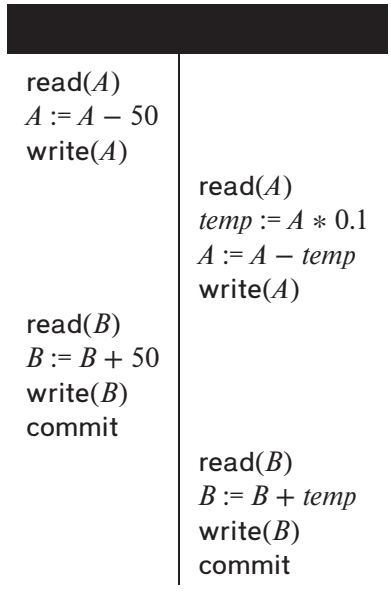


Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 17.5. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution. Indeed, the sum $A + B$ is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

17.6 Serializability

Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider how to determine when a schedule is serializable. Certainly, serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable. Since trans-

Telegram: @uni_k

www.konkur.in

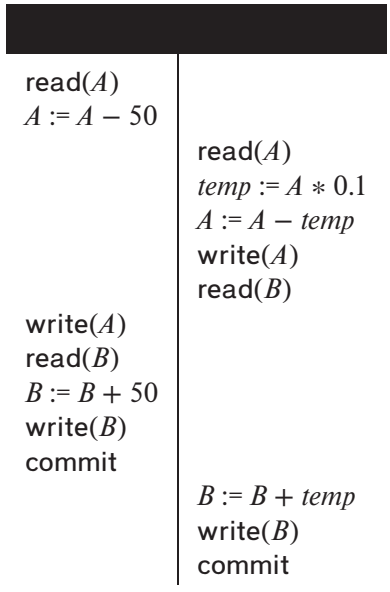


Figure 17.5 Schedule 4—a concurrent schedule resulting in an inconsistent state.

actions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: read and write. We assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. In this model, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. Commit operations, though relevant, are not considered until Section 17.7. We therefore may show only read and write instructions in schedules, as we do for schedule 3 in Figure 17.6.

In this section, we discuss different forms of schedule equivalence but focus on a particular form called **conflict serializability**.

Let us consider a schedule S in which there are two consecutive instructions, I and J, of transactions T_i and T_j, respectively ($i \neq j$). If I and J refer to different data items, then we can swap I and J without affecting the results of any instruction in the schedule. However, if I and J refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. I = read(Q), J = read(Q). The order of I and J does not matter, since the same value of Q is read by T_i and T_j, regardless of the order.

Telegram: @uni_k

www.konkur.in

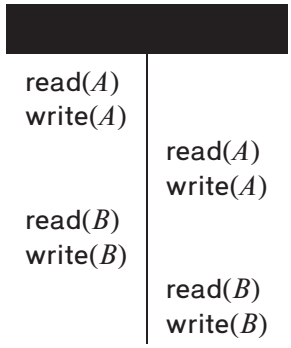


Figure 17.6 Schedule 3—showing only the read and write instructions.

2. I = read(Q), J = write(Q). If I comes before J, then T_i does not read the value of Q that is written by T_j in instruction J. If J comes before I, then T_i reads the value of Q that is written by T_j. Thus, the order of I and J matters.
3. I = write(Q), J = read(Q). The order of I and J matters for reasons similar to those of the previous case.
4. I = write(Q), J = write(Q). Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j. However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both I and J are read instructions does the relative order of their execution not matter.

We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3 in Figure 17.6. The write(A) instruction of T_1 conflicts with the read(A) instruction of T_2. However, the write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1 because the two instructions access different data items.

Let I and J be consecutive instructions of a schedule S. If I and J are instructions of different transactions and I and J do not conflict, then we can swap the order of I and J to produce a new schedule S'. S is equivalent to S', since all instructions appear in the same order in both schedules except for I and J, whose order does not matter.

Since the write(A) instruction of T_2 in schedule 3 of Figure 17.6 does not conflict with the read(B) instruction of T_1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 17.7. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

Telegram: @uni_k

www.konkur.in

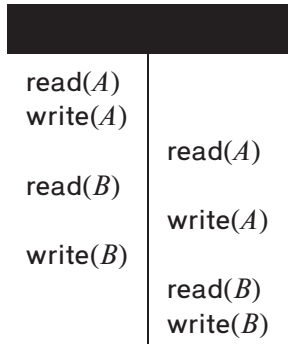


Figure 17.7 Schedule 5—schedule 3 after swapping of a pair of instructions.

We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T_1 with the read(A) instruction of T_2.
- Swap the write(B) instruction of T_1 with the write(A) instruction of T_2.
- Swap the write(B) instruction of T_1 with the read(A) instruction of T_2.

The final result of these swaps, schedule 6 of Figure 17.8, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the read and write instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 produces the same final state as some serial schedule.

If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are **conflict equivalent**.[2]

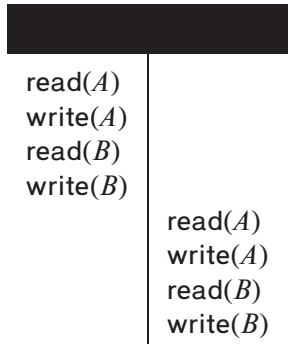


Figure 17.8 Schedule 6—a serial schedule that is equivalent to schedule 3.

[2] We use the term *conflict equivalent* to distinguish the way we have just defined equivalence from other definitions that we shall discuss later on in this section.

Telegram: @uni_k

www.konkur.in

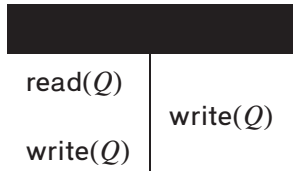


Figure 17.9 Schedule 7.

Not all serial schedules are conflict equivalent to each other. For example, schedules 1 and 2 are not conflict equivalent.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Finally, consider schedule 7 of Figure 17.9; it consists of only the significant operations (that is, the read and write) of transactions T_3 and T_4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule $<T_3, T_4>$ or the serial schedule $<T_4, T_3>$.

We now present a simple and efficient method for determining the conflict serializability of a schedule. Consider a schedule S. We construct a directed graph, called a **precedence graph**, from S. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. T_i executes write(Q) before T_j executes read(Q).
2. T_i executes read(Q) before T_j executes write(Q).
3. T_i executes write(Q) before T_j executes write(Q).

If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S' equivalent to S, T_i must appear before T_j.

For example, the precedence graph for schedule 1 in Figure 17.10a contains the single edge $T_1 \rightarrow T_2$, since all the instructions of T_1 are executed before the first instruction of T_2 is executed. Similarly, Figure 17.10b shows the precedence graph for



Figure 17.10 Precedence graph for (a) schedule 1 and (b) schedule 2.

Telegram: @uni_k

www.konkur.in

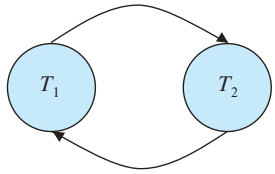


Figure 17.11 Precedence graph for schedule 4.

schedule 2 with the single edge $T_2 \rightarrow T_1$, since all the instructions of T_2 are executed before the first instruction of T_1 is executed.

The precedence graph for schedule 4 appears in Figure 17.11. It contains the edge $T_1 \rightarrow T_2$ because T_1 executes read(A) before T_2 executes write(A). It also contains the edge $T_2 \rightarrow T_1$ because T_2 executes read(B) before T_1 executes write(B).

If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**. There are, in general, several possible linear orders that can be obtained through a topological sort. For example, the graph of Figure 17.12a has the two acceptable linear orderings shown in Figure 17.12b and Figure 17.12c.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms. Cycle-detection algorithms, such as those based on depth-first search, require on the order of n^2 operations, where n is the number of vertices in the graph (that is, the number of transactions).[3]

Returning to our previous examples, note that the precedence graphs for schedules 1 and 2 (Figure 17.10) indeed do not contain cycles. The precedence graph for schedule 4 (Figure 17.11), on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

It is possible to have two schedules that produce the same outcome but that are not conflict equivalent. For example, consider transaction T_5, which transfers \$10 from account B to account A. Let schedule 8 be as defined in Figure 17.13. We claim that schedule 8 is not conflict equivalent to the serial schedule $<T_1, T_5>$, since, in schedule 8, the write(B) instruction of T_5 conflicts with the read(B) instruction of T_1. This creates an edge $T_5 \rightarrow T_1$ in the precedence graph. Similarly, we see that the write(A) instruction of T_1 conflicts with the read instruction of T_5, creating an edge $T_1 \rightarrow T_5$. This shows that the precedence graph has a cycle and that schedule 8 is not serializable. However, the final values of accounts A and B after the execution of either schedule 8 or the serial schedule $<T_1, T_5>$ are the same—\$960 and \$2040, respectively.

[3] If instead we measure complexity in terms of the number of edges, which corresponds to the number of actual conflicts between active transactions, then depth-first-based cycle detection

Telegram: @uni_k

www.konkur.in

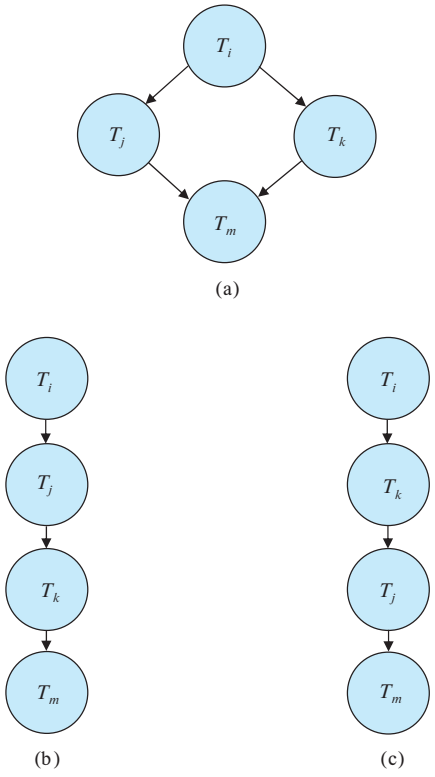


Figure 17.12 Illustration of topological sorting.

We can see from this example that there are less-stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule $<T_1, T_5>$, it must analyze the computation performed by T_1 and T_5, rather than just the read and write operations. In general, such analysis is hard to implement and is computationally expensive. In our example, the final result is the same as that of a serial schedule because of the mathematical fact that the increment and decrement operations are commutative. While this may be easy to see in our simple example, the general case is not so easy since a transaction may be expressed as a complex SQL statement, a Java program with JDBC calls, etc.

However, there are other definitions of schedule equivalence based purely on the read and write operations. One such definition is *view equivalence*, a definition that leads to the concept of *view serializability*. View serializability is not used in practice due to its high degree of computational complexity.[4] We therefore defer discussion of

[4]Testing for view serializability has been proven to be NP-complete, which means that it is virtually certain that no efficient test for view serializability exists.

Telegram: @uni_k

www.konkur.in

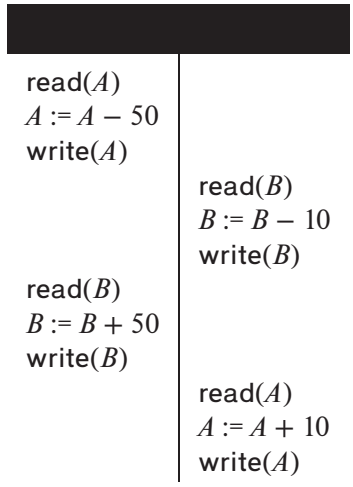


Figure 17.13 Schedule 8.

view serializability to Chapter 18, but, for completeness, note here that the example of schedule 8 is not view serializable.

17.7 Transaction Isolation and Atomicity

So far, we have studied schedules while assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, the atomicity property requires that any transaction T_j that is dependent on T_i (i.e., T_j has read data written by T_i) is also aborted. To achieve this, we need to place restrictions on the types of schedules permitted in the system.

In the following two subsections, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure. We describe in Chapter 18 how to ensure that only such acceptable schedules are generated.

17.7.1 Recoverable Schedules

Consider the partial schedule 9 in Figure 17.14, in which T_7 is a transaction that performs only one instruction: read(A). We call this a **partial schedule** because we have not included a **commit** or **abort** operation for T_6. Notice that T_7 commits immediately after executing the read(A) instruction. Thus, T_7 commits while T_6 is still in the active state. Now suppose that T_6 fails before it commits. T_7 has read the value of data item A written by T_6. Therefore, we say that T_7 is **dependent** on T_6. Because of this, we must abort T_7 to ensure atomicity. However, T_7 has already committed and cannot be

Telegram: @uni_k

www.konkur.in

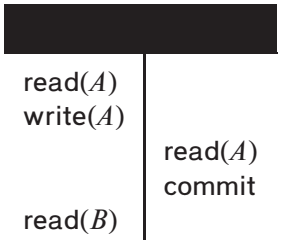


Figure 17.14 Schedule 9, a nonrecoverable schedule.

aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T_6.

Schedule 9 is an example of a *nonrecoverable* schedule. A **recoverable schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j. For the example of schedule 9 to be recoverable, T_7 would have to delay committing until after T_6 commits.

17.7.2 Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction T_i, we may have to roll back several transactions. Such situations occur if transactions have read data written by T_i. As an illustration, consider the partial schedule of Figure 17.15. Transaction T_8 writes a value of A that is read by transaction T_9. Transaction T_9 writes a value of A that is read by transaction T_{10}. Suppose that, at this point, T_8 fails. T_8 must be rolled back. Since T_9 is dependent on T_8, T_9 must be rolled back. Since T_{10} is dependent on T_9, T_{10} must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.

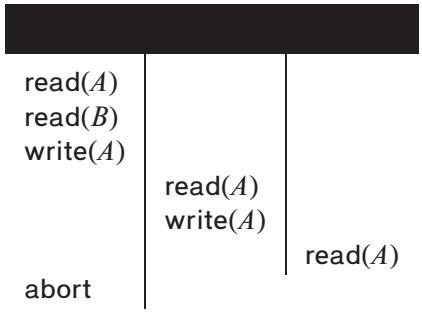


Figure 17.15 Schedule 10.

Telegram: @uni_k

www.konkur.in

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j. It is easy to verify that every cascadeless schedule is also recoverable.

17.8 Transaction Isolation Levels

Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions. If every transaction has the property that it maintains database consistency if executed alone, then serializability ensures that concurrent executions maintain consistency. However, the protocols required to ensure serializability may allow too little concurrency for certain applications. In these cases, weaker levels of consistency are used. The use of weaker levels of consistency places additional burdens on programmers for ensuring database correctness.

The SQL standard also allows a transaction to specify that it may be executed in such a way that it becomes nonserializable with respect to other transactions. For instance, a transaction may operate at the isolation level of **read uncommitted**, which permits the transaction to read a data item even if it was written by a transaction that has not been committed. SQL provides such features for the benefit of long transactions whose results do not need to be precise. If these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed.

The **isolation levels** specified by the SQL standard are as follows:

- **Serializable** usually ensures serializable execution. However, as we shall explain shortly, some database systems implement this isolation level in a manner that may, in certain cases, allow nonserializable executions.
- **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction.
- **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
- **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

Telegram: @uni_k

www.konkur.in

All the isolation levels above additionally disallow **dirty writes**, that is, they disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted.

Many database systems run, by default, at the read-committed isolation level. In SQL, it is possible to set the isolation level explicitly, rather than accepting the system's default setting. For example, the statement

set transaction isolation level serializable

sets the isolation level to serializable; any of the other isolation levels may be specified instead. The preceding syntax is supported by Oracle, PostgreSQL, and SQL Server; Oracle uses the syntax

alter session set isolation_level = serializable

while DB2 uses the syntax "**change isolation level**" with its own abbreviations for isolation levels. Changing of the isolation level must be done as the first statement of a transaction.

By default, most databases commit individual statements as soon as they are executed. Such **automatic commit** of individual statements must be turned off to allow multiple statements to run as a single transaction. The command **start transaction** ensures that subsequent SQL statements, until a subsequent **commit** or **rollback**, are executed as a single transaction. As expected, the **commit** operation commits the preceding SQL statements, while **rollback** rolls back the preceding SQL statements. (SQL Server uses **begin transaction** in place of **start transaction**, while Oracle and PostgreSQL treat **begin** as identical to **start transaction**.)

APIs such as JDBC and ODBC provide functions to turn off automatic commit. In JDBC the setAutoCommit method of the Connection interface (which we saw earlier in Section 5.1.1.8) can be used to turn automatic commit off by invoking setAutoCommit(false), or on by invoking setAutoCommit(true). Further, in JDBC the method setTransactionIsolation(int level) of the Connection interface can be invoked with any one of

- Connection.TRANSACTION_SERIALIZABLE,
- Connection.TRANSACTION_REPEATABLE_READ,
- Connection.TRANSACTION_READ_COMMITTED, or
- Connection.TRANSACTION_READ_UNCOMMITTED

to set the transaction isolation level correspondingly.

An application designer may decide to accept a weaker isolation level in order to improve system performance. As we shall see in Section 17.9 and Chapter 18, ensuring serializability may force a transaction to wait for other transactions or, in some cases, to abort because the transaction can no longer be executed as part of a serializable execution. While it may seem shortsighted to risk database consistency for performance,

Telegram: @uni_k

www.konkur.in

this trade-off makes sense if we can be sure that the inconsistency that may occur is not relevant to the application.

There are many means of implementing isolation levels. As long as the implementation ensures serializability, the designer of a database application or a user of an application does not need to know the details of such implementations, except perhaps for dealing with performance issues. Unfortunately, even if the isolation level is set to **serializable**, some database systems actually implement a weaker level of isolation, which does not rule out every possible nonserializable execution; we revisit this issue in Section 17.9. If weaker levels of isolation are used, either explicitly or implicitly, the application designer has to be aware of some details of the implementation, to avoid or minimize the chance of inconsistency due to lack of serializability.

17.9 Implementation of Isolation Levels

So far, we have seen what properties a schedule must have if it is to leave the database in a consistent state and allow transaction failures to be handled in a safe manner.

There are various **concurrency-control** policies that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating system time-shares resources (such as CPU time) among the transactions.

As a trivial example of a concurrency-control policy, consider this: A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are recoverable and cascadeless as well.

A concurrency-control policy such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In other words, it provides a poor degree of concurrency (indeed, no concurrency at all). As we saw in Section 17.5, concurrent execution has substantial performance benefits.

The goal of concurrency-control policies is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, recoverable, and cascadeless.

Here we provide an overview of how some of most important concurrency-control mechanisms work, and we defer the details to Chapter 18.

17.9.1 Locking

Instead of locking the entire database, a transaction could instead lock only those data items that it accesses. Under such a policy, the transaction must hold locks long enough to ensure serializability, but for a period short enough not to harm performance exces-

Telegram: @uni_k

www.konkur.in



sively. Complicating matters are SQL statements where the data items accessed depend on a **where** clause, which we discuss in Section 17.10. In Chapter 18, we present the two-phase locking protocol, a simple, widely used technique that ensures serializability. Stated simply, two-phase locking requires a transaction to have two phases, one where it acquires locks but does not release any, and a second phase where the transaction releases locks but does not acquire any. (In practice, locks are usually released only when the transaction completes its execution and has been either committed or aborted.)

Telegram: @uni_k

www.konkur.in

Further improvements to locking result if we have two kinds of locks: shared and exclusive. Shared locks are used for data that the transaction reads and exclusive locks are used for those it writes. Many transactions can hold shared locks on the same data item at the same time, but a transaction is allowed an exclusive lock on a data item only if no other transaction holds any lock (regardless of whether shared or exclusive) on the data item. This use of two modes of locks along with two-phase locking allows concurrent reading of data while still ensuring serializability.

17.9.2 Timestamps

Another category of techniques for the implementation of isolation assigns each transaction a **timestamp**, typically when it begins. For each data item, the system keeps two timestamps. The read timestamp of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item. The write timestamp of a data item holds the timestamp of the transaction that wrote the current value of the data item. Timestamps are used to ensure that transactions access each data item in order of the transactions' timestamps if their accesses conflict. When this is not possible, offending transactions are aborted and restarted with a new timestamp.

17.9.3 Multiple Versions and Snapshot Isolation

By maintaining more than one version of a data item, it is possible to allow a transaction to read an old version of a data item rather than a newer version written by an uncommitted transaction or by a transaction that should come later in the serialization order. There are a variety of multiversion concurrency-control techniques. One in particular, called **snapshot isolation**, is widely used in practice.

In snapshot isolation, we can imagine that each transaction is given its own version, or snapshot, of the database when it begins.[5] It reads data from this private version and is thus isolated from the updates made by other transactions. If the transaction updates the database, that update appears only in its own version, not in the actual database itself. Information about these updates is saved so that the updates can be applied to the "real" database if the transaction commits.

When a transaction T enters the partially committed state, it then proceeds to the committed state only if no other concurrent transaction has modified data that T intends to update. Transactions that, as a result, cannot commit abort instead.

Snapshot isolation ensures that attempts to read data never need to wait (unlike locking). Read-only transactions cannot be aborted; only those that modify data run a slight risk of aborting. Since each transaction reads its own version or snapshot of the database, reading data does not cause subsequent update attempts by other transactions to wait (unlike locking). Since most transactions are read-only (and most others read more data than they update), this is often a major source of performance improvement as compared to locking.

[5] In reality, the entire database is not copied. Multiple versions are

Telegram: @uni_k

www.konkur.in

The problem with snapshot isolation is that, paradoxically, it provides *too much* isolation. Consider two transactions T and T'. In a serializable execution, either T sees all the updates made by T' or T' sees all the updates made by T, because one must follow the other in the serialization order. Under snapshot isolation, there are cases where neither transaction sees the updates of the other. This is a situation that cannot occur in a serializable execution. In many (indeed, most) cases, the data accesses by the two transactions do not conflict and there is no problem. However, if T reads some data item that T' updates and T' reads some data item that T updates, it is possible that both transactions fail to read the update made by the other. The result, as we shall see in Chapter 18, may be an inconsistent database state that, of course, could not be obtained in any serializable execution.

Oracle, PostgreSQL, and SQL Server offer the option of snapshot isolation. Oracle and PostgreSQL versions prior to PostgreSQL 9.1 implement the **serializable** isolation level using snapshot isolation. As a result, their implementation of serializability can, in exceptional circumstances, result in a nonserializable execution being allowed. SQL Server instead includes an additional isolation level beyond the standard ones, called **snapshot**, to offer the option of snapshot isolation. PostgreSQL versions subsequent to 9.1 implement a form of concurrency control called serializable snapshot isolation, which provides the benefits of snapshot isolation while ensuring serializability.

17.10 Transactions as SQL Statements

In Section 4.3, we presented the SQL syntax for specifying the beginning and end of transactions. Now that we have seen some of the issues in ensuring the ACID properties for transactions, we are ready to consider how those properties are ensured when transactions are specified as a sequence of SQL statements rather than the restricted model of simple reads and writes that we considered up to this point.

In our simple model, we assumed a set of data items exists. While our simple model allowed data-item values to be changed, it did not allow data items to be created or deleted. In SQL, however, **insert** statements create new data and **delete** statements delete data. These two statements are, in effect, **write** operations, since they change the database, but their interactions with the actions of other transactions are different from what we saw in our simple model. As an example, consider how insertion or deletion would conflict with the following SQL query, which finds all instructors who earn more than $90,000:

select *ID*, *name*
from *instructor*
where *salary* > 90000;

Using our sample *instructor* relation (Section A.3), we find that only Einstein and Brandt satisfy the condition. Now assume that around the same time we are running our query, another user inserts a new instructor named “James” whose salary is $100,000.

Telegram: @uni_k

www.konkur.in

insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000);

The result of our query depends on whether this insert comes before or after our query is run. In a concurrent execution of these transactions, it is intuitively clear that they conflict, but this is a conflict that may not be captured by our simple model. This situation is referred to as the **phantom phenomenon** because a conflict may exist on "phantom" data.

Our simple model of transactions required that operations operate on a specific data item given as an argument to the operation. In our simple model, we can look at the **read** and **write** steps to see which data items are referenced. But in an SQL statement, the specific data items (tuples) referenced may be determined by a **where** clause predicate. So the same transaction, if run more than once, might reference different data items each time it is run if the values in the database change between runs. In our example, the 'James' tuple is referenced only if our query comes after the insertion. Let T denote the query and let T' denote the insert. If T' comes first, then there is an edge $T' \rightarrow T$ in the precedence graph. However, in the case where the query T comes first, there is no edge in the precedence graph between T and T' despite the actual conflict on phantom data that forces T to be serialized before T'.

The above-mentioned problem demonstrates that it is not sufficient for concurrency control to consider only the tuples that are accessed by a transaction; the information used to find the tuples that are accessed by the transaction must also be considered for the purpose of concurrency control. The information used to find tuples could be updated by an insertion or deletion, or in the case of an index, even by an update to a search-key attribute. For example, if locking is used for concurrency control, the data structures that track the tuples in a relation, as well as index structures, must be appropriately locked. However, such locking can lead to poor concurrency in some situations; index-locking protocols that maximize concurrency, while ensuring serializability in spite of inserts, deletes, and predicates in queries, are discussed in Section 18.4.3.

Let us consider again the query:

select *ID*, *name*
from *instructor*
where *salary*> 90000;

and the following SQL update:

update *instructor*
set *salary* = *salary* * 0.9
where *name* = 'Wu';

We now face an interesting situation in determining whether our query conflicts with the update statement. If our query reads the entire *instructor* relation, then it reads the

Telegram: @uni_k

www.konkur.in

tuple with Wu's data and conflicts with the update. However, if an index were available that allowed our query direct access to those tuples with *salary* > 90000, then our query would not have accessed Wu's data at all because Wu's salary is initially $90,000 in our example instructor relation and reduces to $81,000 after the update.

However, using the above approach, it would appear that the existence of a conflict depends on a low-level query processing decision by the system that is unrelated to a user-level view of the meaning of the two SQL statements! An alternative approach to concurrency control treats an insert, delete, or update as conflicting with a predicate on a relation, if it could affect the set of tuples selected by a predicate. In our example query above, the predicate is "*salary* > 90000", and an update of Wu's salary from $90,000 to a value greater than $90,000, or an update of Einstein's salary from a value greater than $90,000 to a value less than or equal to $90,000, would conflict with this predicate. Locking based on this idea is called **predicate locking**; predicate locking is often implemented using locks on index nodes as we see in Section 18.4.3.

17.11 Summary

- A *transaction* is a *unit* of program execution that accesses and possibly updates various data items. Understanding the concept of a transaction is critical for understanding and implementing updates of data in a database in such a way that concurrent executions and failures of various forms do not result in the database becoming inconsistent.
- Transactions are required to have the ACID properties: atomicity, consistency, isolation, and durability.
  - Atomicity ensures that either all the effects of a transaction are reflected in the database, or none are; a failure cannot leave the database in a state where a transaction is partially executed.
  - Consistency ensures that, if the database is initially consistent, the execution of the transaction (by itself) leaves the database in a consistent state.
  - Isolation ensures that concurrently executing transactions are isolated from one another, so that each has the impression that no other transaction is executing concurrently with it.
  - Durability ensures that, once a transaction has been committed, that transaction's updates do not get lost, even if there is a system failure.
- Concurrent execution of transactions improves throughput of transactions and system utilization and also reduces the waiting time of transactions.
- The various types of storage in a computer are volatile storage, non-volatile storage, and stable storage. Data in volatile storage, such as in RAM, are lost when the computer crashes. Data in non-volatile storage, such as disk, are not lost when

Telegram: @uni_k

www.konkur.in

the computer crashes but may occasionally be lost because of failures such as disk crashes. Data in stable storage are never lost.

- Stable storage that must be accessible online is approximated with mirrored disks, or other forms of RAID, which provide redundant data storage. Offline, or archival, stable storage may consist of multiple tape copies of data stored in physically secure locations.
- When several transactions execute concurrently on the database, the consistency of data may no longer be preserved. It is therefore necessary for the system to control the interaction among the concurrent transactions.
    - Since a transaction is a unit that preserves consistency, a serial execution of transactions guarantees that consistency is preserved.
    - A *schedule* captures the key actions of transactions that affect concurrent execution, such as `read` and `write` operations, while abstracting away internal details of the execution of the transaction.
    - We require that any schedule produced by concurrent processing of a set of transactions will have an effect equivalent to a schedule produced when these transactions are run serially in some order.
    - A system that guarantees this property is said to ensure *serializability*.
    - There are several different notions of equivalence leading to the concepts of *conflict serializability* and *view serializability*.
- Serializability of schedules generated by concurrently executing transactions can be ensured through one of a variety of mechanisms called *concurrency-control* policies.
- We can test a given schedule for conflict serializability by constructing a *precedence graph* for the schedule and by searching for the absence of cycles in the graph. However, there are more efficient concurrency-control policies for ensuring serializability.
- Schedules must be recoverable, to make sure that if transaction a sees the effects of transaction b, and b then aborts, then a also gets aborted.
- Schedules should preferably be cascadeless, so that the abort of a transaction does not result in cascading aborts of other transactions. Cascadelessness is ensured by allowing transactions to only read committed data.
- The concurrency-control management component of the database is responsible for handling the concurrency-control policies. Techniques include locking, timestamp ordering, and snapshot isolation. Chapter 18 describes concurrency-control policies.

Telegram: @uni_k

www.konkur.in

- Database systems offer isolation levels weaker than serializability to allow less restriction of concurrency and thus improved performance. This introduces a risk of inconsistency that some applications find acceptable.
- Ensuring correct concurrent execution in the presence of SQL **update**, **insert**, and **delete** operations requires additional care due to the phantom phenomenon.

Review Terms

- Transaction
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Inconsistent state
- Storage types
  - Volatile storage
  - Non-volatile storage
  - Stable storage
- Concurrency-control system
- Recovery system
- Transaction state
  - Active
  - Partially committed
  - Failed
  - Aborted
  - Committed
  - Terminated
- compensating transaction
- Transaction
  - Restart
  - Kill
- Observable external writes
- Concurrent executions
- Serial execution
- Schedules
- Conflict of operations
- Conflict equivalence
- Conflict serializability
- Serializability testing
- Precedence graph
- Serializability order
- Recoverable schedules
- Cascading rollback
- Cascadeless schedules
- Isolation levels
  - Serializable
  - Repeatable read
  - Read committed
  - Read uncommitted
- Dirty writes
- Automatic commit
- Concurrency control
- Locking
- Timestamp ordering
- Snapshot isolation
- Phantom phenomenon
- Predicate locking

Telegram: @uni_k

www.konkur.in

Practice Exercises

17.1 Suppose that there is a database system that never fails. Is a recovery manager required for this system?

17.2 Consider a file system such as the one on your favorite operating system.

a. What are the steps involved in the creation and deletion of files and in writing data to a file?

b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files and to writing data to files.

17.3 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

17.4 What class or classes of storage can be used to ensure durability? Why?

17.5 Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

17.6 Consider the precedence graph of Figure 17.16. Is the corresponding schedule conflict serializable? Explain your answer.

17.7 What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be desirable to allow noncascadeless schedules? Explain your answer.

17.8 The **lost update** anomaly is said to occur if a transaction T_j reads a data item, then another transaction T_k writes the data item (possibly based on a previous read), after which T_j writes the data item. The update performed by T_k has been lost, since the update done by T_j ignored the value written by T_k.

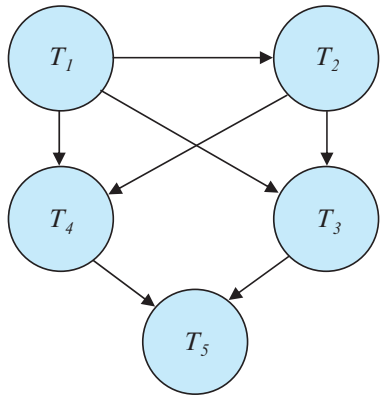


Figure 17.16 Precedence graph for Practice Exercise 17.6.

Telegram: @uni_k

www.konkur.in

a. Give an example of a schedule showing the lost update anomaly.

b. Give an example schedule to show that the lost update anomaly is possible with the **read committed** isolation level.

c. Explain why the lost update anomaly is not possible with the **repeatable read** isolation level.

17.9 Consider a database for a bank where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs that would present a problem for the bank.

17.10 Consider a database for an airline where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs, but the airline may be willing to accept it in order to gain better overall performance.

17.11 The definition of a schedule assumes that operations can be totally ordered by time. Consider a database system that runs on a system with multiple processors, where it is not always possible to establish an exact ordering between operations that executed on different processors. However, operations on a data item can be totally ordered.

Does this situation cause any problem for the definition of conflict serializability? Explain your answer.

Exercises

17.12 List the ACID properties. Explain the usefulness of each.

17.13 During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.

17.14 Explain the distinction between the terms *serial schedule* and *serializable schedule*.

17.15 Consider the following two transactions:

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{14}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Telegram: @uni_k

www.konkur.in

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ as the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of T_{13} and T_{14} that produces a nonserializable schedule.

c. Is there a concurrent execution of T_{13} and T_{14} that produces a serializable schedule?

17.16 Give an example of a serializable schedule with two transactions such that the order in which the transactions commit is different from the serialization order.

17.17 What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow nonrecoverable schedules? Explain your answer.

17.18 Why do database systems support concurrent execution of transactions, despite the extra effort needed to ensure that concurrent execution does not cause any problems?

17.19 Explain why the read-committed isolation level ensures that schedules are cascade-free.

17.20 For each of the following isolation levels, give an example of a schedule that respects the specified level of isolation but is not serializable:

a. Read uncommitted

b. Read committed

c. Repeatable read

17.21 Suppose that in addition to the operations read and write, we allow an operation pred_read(r, P), which reads all tuples in relation r that satisfy predicate P.

a. Give an example of a schedule using the pred_read operation that exhibits the phantom phenomenon and is nonserializable as a result.

b. Give an example of a schedule where one transaction uses the pred_read operation on relation r and another concurrent transaction deletes a tuple from r, but the schedule does not exhibit a phantom conflict. (To do so, you have to give the schema of relation r and show the attribute values of the deleted tuple.)

Telegram: @uni_k

www.konkur.in

Further Reading

[Gray and Reuter (1993)] provides detailed textbook coverage of transaction-processing concepts, techniques, and implementation details, including concurrency control and recovery issues. [Bernstein and Newcomer (2009)] provides textbook coverage of various aspects of transaction processing.

The concept of serializability was formalized by [Eswaran et al. (1976)] in connection with work on concurrency control for System R.

References covering specific aspects of transaction processing, such as concurrency control and recovery, are cited in Chapter 18 and Chapter 19.

Bibliography

[Bernstein and Newcomer (2009)] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd edition, Morgan Kaufmann (2009).

[Eswaran et al. (1976)] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Volume 19, Number 11 (1976), pages 624–633.

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 18



Concurrency Control

We saw in Chapter 17 that one of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes. In this chapter, we consider the management of concurrently executing transactions, and we ignore failures. In Chapter 19, we shall see how the system can recover from failures.

As we shall see, there are a variety of concurrency-control schemes. No one scheme is clearly the best; each one has advantages. In practice, the most frequently used schemes are *two-phase locking* and *snapshot isolation*.

18.1 Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item. We introduced the concept of locking in Section 17.9.

18.1.1 Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. **Shared**. If a transaction T_i has obtained a **shared-mode lock** (denoted by S) on item Q, then T_i can read, but cannot write, Q.
2. **Exclusive**. If a transaction T_i has obtained an **exclusive-mode lock** (denoted by X) on item Q, then T_i can both read and write Q.

We require that every transaction **request** a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction

Telegram: @uni_k

www.konkur.in

	S	X
S	true	false
X	false	false

Figure 18.1 Lock-compatibility matrix comp.

makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

To state this more generally, given a set of lock modes, we can define a **compatibility function** on them as follows: Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is **compatible** with mode B. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure 18.1. An element comp(A, B) of the matrix has the value *true* if and only if mode A is compatible with mode B.

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction. A transaction can unlock a data item Q by the unlock(Q) instruction.

To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to **wait** until all incompatible locks held by other transactions have been released.

Transaction T_i may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

As an illustration, consider again the banking example that we introduced in Chapter 17. Let A and B be two accounts that are accessed by transactions T_1 and T_2. Transaction T_1 transfers \$50 from account B to account A (Figure 18.2). Transaction T_2 displays the total amount of money in accounts A and B—that is, the sum $A + B$ (Figure 18.3).

Telegram: @uni_k

www.konkur.in


```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 18.2 Transaction T_1.

Suppose that the values of accounts A and B are \$100 and \$200, respectively. If these two transactions are executed serially, either in the order T_1, T_2 or the order T_2, T_1, then transaction T_2 will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 18.4, is possible. In this case, transaction T_2 displays \$250, which is incorrect. The reason for this mistake is that the transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency-control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction. Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction. We shall therefore drop the

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 18.3 Transaction T .

Telegram: @uni_k

www.konkur.in

lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 18.4 Schedule 1.

column depicting the actions of the concurrency-control manager from all schedules depicted in the rest of the chapter. We let you infer when locks are granted.

Suppose now that unlocking is delayed to the end of the transaction. Transaction T_3 corresponds to T_1 with unlocking delayed (Figure 18.5). Transaction T_4 corresponds to T_2 with unlocking delayed (Figure 18.6).

You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of $250 being displayed, is no longer possible with T_3 and T_4. Other schedules are possible. T_4 will not print out an inconsistent result in any of them; we shall see why later.

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 18.7 for T_3 and T_4. Since T_3 is holding an exclusive-mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**. When deadlock occurs, the system must roll back one of

Telegram: @uni_k

www.konkur.in


```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

Figure 18.5 Transaction T_3 (transaction T_1 with unlocking delayed).

the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution. We shall return to the issue of deadlock handling in Section 18.2.

If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations, as we shall see in Section 18.1.5. However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Figure 18.6 Transaction T_4 (transaction T with unlocking delayed).

Telegram: @uni_k

www.konkur.in

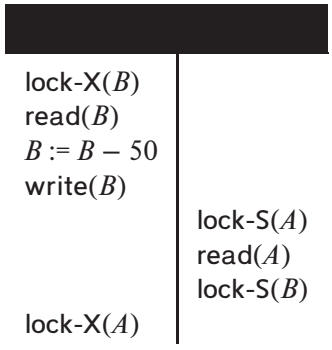


Figure 18.7 Schedule 2.

schedules is a proper subset of all possible serializable schedules. We shall present several locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation. Before doing so, we introduce some terminology.

Let $\{T_0, T_1, \ldots, T_n\}$ be a set of transactions participating in a schedule S. We say that T_i **precedes** T_j in S, written $T_i \rightarrow T_j$, if there exists a data item Q such that T_i has held lock mode A on Q, and T_j has held lock mode B on Q later, and comp(A,B) = false. If $T_i \rightarrow T_j$, then that precedence implies that in any equivalent serial schedule, T_i must appear before T_j. Observe that this graph is similar to the precedence graph that we used in Section 17.6 to test for conflict serializability. Conflicts between instructions correspond to noncompatibility of lock modes.

We say that a schedule S is **legal** under a given locking protocol if S is a possible schedule for a set of transactions that follows the rules of the locking protocol. We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable; in other words, for all legal schedules the associated $\rightarrow$ relation is acyclic.

18.1.2 Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it

Telegram: @uni_k

www.konkur.in

is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved**.

We can avoid starvation of transactions by granting locks in the following manner: When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:

- There is no other transaction holding a lock on Q in a mode that conflicts with M.
- There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i.

Thus, a lock request will never get blocked by a lock request that is made later.

18.1.3 The Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions T_3 and T_4 are two phase. On the other hand, transactions T_1 and T_2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction T_3, we could move the unlock(B) instruction to just after the lock-X(A) instruction and still retain the two-phase locking property.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do (see Practice Exercise 18.1).

Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions T_3 and T_4 are two phase, but, in schedule 2 (Figure 18.7), they are deadlocked.

Recall from Section 17.7.2 that, in addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 18.8. Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.

Telegram: @uni_k

www.konkur.in

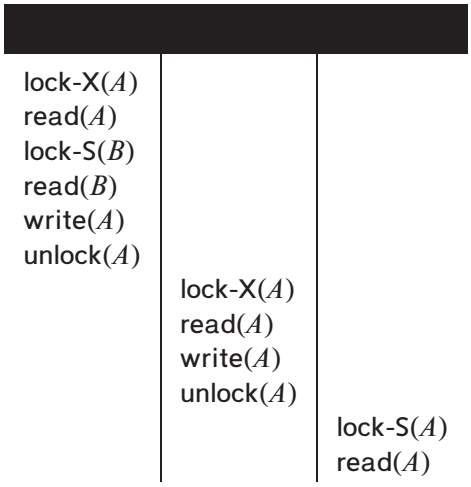


Figure 18.8 Partial schedule under two-phase locking.

Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

If we employ the two-phase locking protocol, then T_8 must lock a_1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that T_8 needs an exclusive lock on a only at the end of

Telegram: @uni_k

www.konkur.in

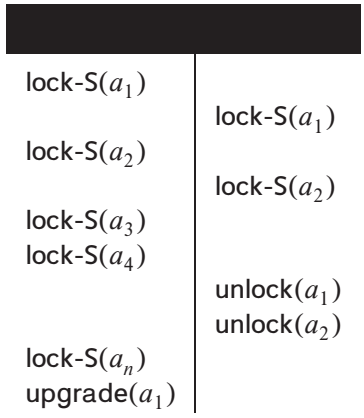


Figure 18.9 Incomplete schedule with a lock conversion.

its execution, when it writes a_1. Thus, if T_8 could initially lock a_1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since T_8 and T_9 could access a_1 and a_2 simultaneously.

This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Returning to our example, transactions T_8 and T_9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 18.9, where only some of the locking instructions are shown.

Note that a transaction attempting to upgrade a lock on an item Q may be forced to wait. This enforced wait occurs if Q is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. We shall see examples when we consider other locking protocols later in this chapter.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database

Telegram: @uni_k

www.konkur.in

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.
- When T_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

18.1.4 Implementation of Locking

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

Figure 18.10 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7 and is waiting for a lock on I4.

Although the figure does not show it, the lock table should also maintain an index on transaction identifiers so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

Telegram: @uni_k

www.konkur.in

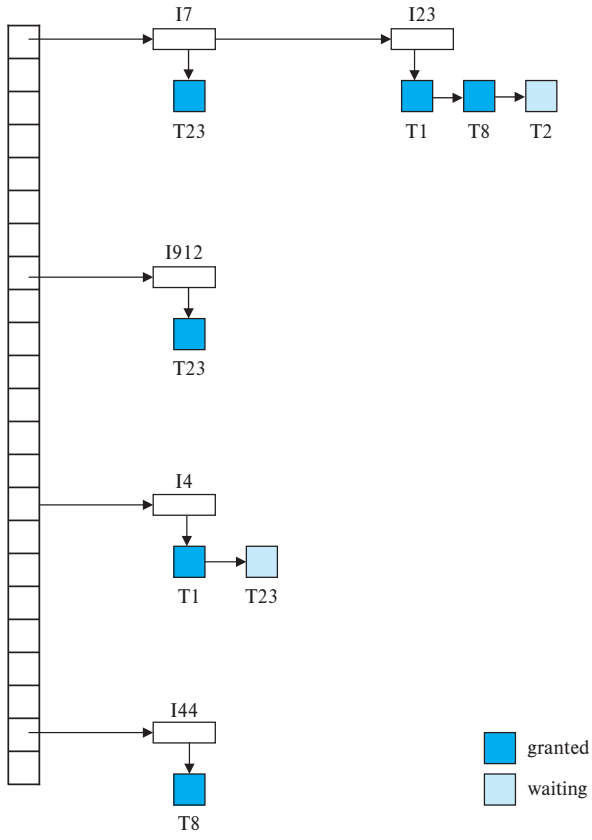


Figure 18.10 Lock table.

It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request and processes the record following it, if any, similarly, and so on.
- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction (see Section 19.3), it releases all locks held by the aborted transaction.

Telegram: @uni_k

www.konkur.in

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted. We study how to detect and handle deadlocks later, in Section 18.2.2. Section 20.3.1 describes an alternative implementation—one that uses shared memory instead of message passing for lock request/grant.

18.1.5 Graph-Based Protocols

As noted in Section 18.1.3, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j. This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We shall present a simple protocol, called the *tree protocol*, which is restricted to employ only *exclusive* locks. References to other, more complex, graph-based locking protocols are in the online bibliographical notes.

In the **tree protocol**, the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of Figure 18.11. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

Telegram: @uni_k

www.konkur.in

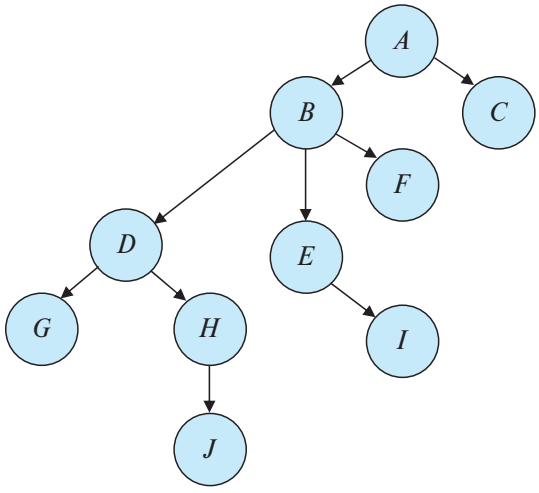


Figure 18.11 Tree-structured database graph.

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears in Figure 18.12. Note that, during its execution, transaction T_{10} holds locks on two *disjoint* subtrees.

Observe that the schedule of Figure 18.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.

The tree protocol in Figure 18.12 does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write, we record which transaction performed the last write to the data item. Whenever a transaction T_i performs a read of an uncommitted data item, we record a **commit dependency** of T_i on the transaction that performed the last write to the data item. Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, T_i must also be aborted.

Telegram: @uni_k

www.konkur.in

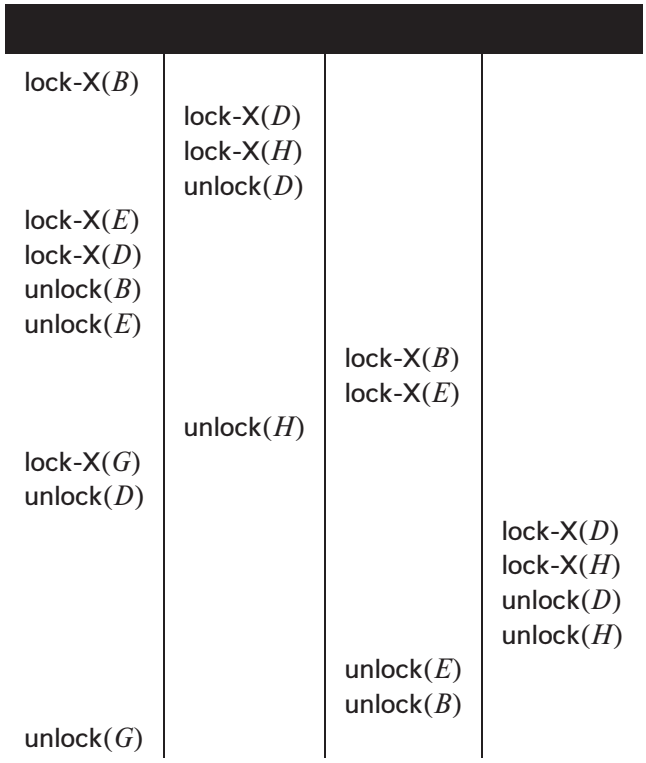


Figure 18.12 Serializable schedule under the tree protocol.

The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items A and J in the database graph of Figure 18.11 must lock not only A and J, but also data items B, D, and H. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa. Examples of such schedules are explored in the exercises.

Telegram: @uni_k

www.konkur.in

18.2 Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and … , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

18.2.1 Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and it performs transaction rollback instead of waiting for a lock whenever the wait could potentially result in a deadlock.

The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol: (1) it is often hard to predict, before the transaction begins, what data items need to be locked; (2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Another approach for preventing deadlocks is to impose an ordering of all data items and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement,

Telegram: @uni_k

www.konkur.in

as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be **preempted** by rolling back of T_i, and granting of the lock to T_j. To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlock-prevention schemes using timestamps have been proposed:

1. The **wait–die** scheme is a nonpreemptive technique. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j (i.e., T_i is older than T_j). Otherwise, T_i is rolled back (dies).

   For example, suppose that transactions T_{14}, T_{15}, and T_{16} have timestamps 5, 10, and 15, respectively. If T_{14} requests a data item held by T_{15}, then T_{14} will wait. If T_{16} requests a data item held by T_{15}, then T_{16} will be rolled back.

2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j (i.e., T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is *wounded* by T_i).

   Returning to our example, with transactions T_{14}, T_{15}, and T_{16}, if T_{14} requests a data item held by T_{15}, then the data item will be preempted from T_{15}, and T_{15} will be rolled back. If T_{16} requests a data item held by T_{15}, then T_{16} will wait.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

Another simple approach to deadlock prevention is based on **lock timeouts**. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery, which Section 18.2.2 discusses.

The timeout scheme is particularly easy to implement, and it works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.

Telegram: @uni_k

www.konkur.in

Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

18.2.2 Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

In this section, we elaborate on these issues.

18.2.2.1 Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$. If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.

When transaction T_i requests a data item currently being held by transaction T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the wait-for graph in Figure 18.13, which depicts the following situation:

- Transaction T_{17} is waiting for transactions T_{18} and T_{19}.
- Transaction T_{19} is waiting for transaction T_{18}.
- Transaction T_{18} is waiting for transaction T .

Telegram: @uni_k

www.konkur.in

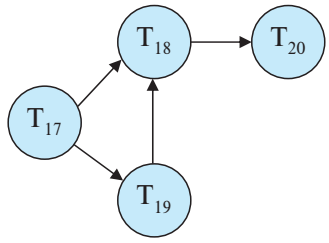


Figure 18.13 Wait-for graph with no cycle.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction T_{20} is requesting an item held by T_{19}. The edge $T_{20} \rightarrow T_{19}$ is added to the wait-for graph, resulting in the new system state in Figure 18.14. This time, the graph contains the cycle:

$$T_{18} \rightarrow T_{20} \rightarrow T_{19} \rightarrow T_{18}$$

implying that transactions T_{18}, T_{19}, and T_{20} are all deadlocked.

Consequently, the question arises: When should we invoke the detection algorithm? The answer depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

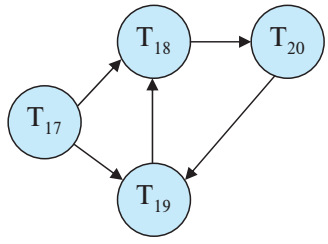


Figure 18.14 Wait-for graph with a cycle.

Telegram: @uni_k

www.konkur.in

18.2.2.2 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim**. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:
   a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b. How many data items the transaction has used.
   c. How many more data items the transaction needs for it to complete.
   d. How many transactions will be involved in the rollback.
2. **Rollback**. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the online bibliographical notes for relevant references.
3. **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

18.3 Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization

Telegram: @uni_k

www.konkur.in

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction T_i needs to access an entire relation, and a locking protocol is used to lock tuples, then T_i must lock each tuple in the relation. Clearly, acquiring many such locks is time-consuming; even worse, the lock table may become very large and no longer fit in memory. It would be better if T_i could issue a *single* lock request to lock the entire relation. On the other hand, if transaction T_j needs to access only a few tuples, it should not be required to lock the entire relation, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of **granularity**. This is done by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol (Section 18.1.5). A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of Figure 18.15, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction T_i gets an **explicit lock** on file F_c of Figure 18.15, in exclusive mode, then it has an **implicit**

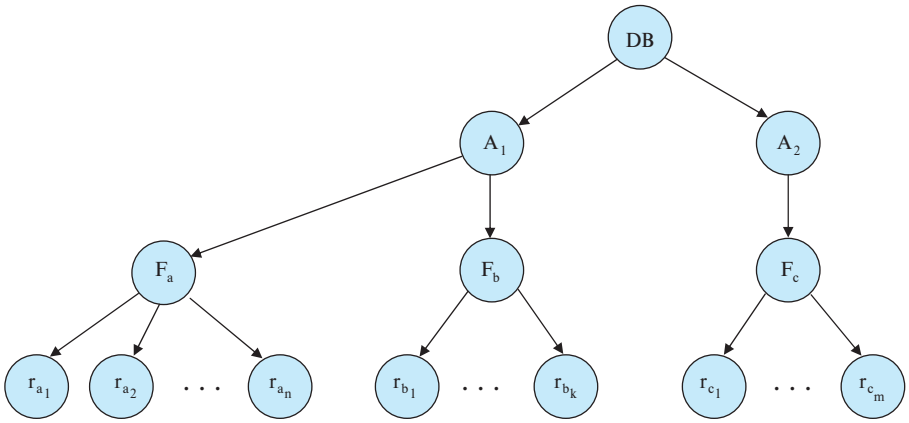


Figure 18.15 Granularity hierarchy.

Telegram: @uni_k

www.konkur.in

lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of F_c explicitly.

Suppose that transaction T_j wishes to lock record r_{b_6} of file F_b. Since T_i has locked F_b explicitly, it follows that r_{b_6} is also locked (implicitly). But, when T_j issues a lock request for r_{b_6}, r_{b_6} is not explicitly locked! How does the system determine whether T_j can lock r_{b_6}? T_j must traverse the tree from the root to record r_{b_6}. If any node in that path is locked in an incompatible mode, then T_j must be delayed.

Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node–say, Q–must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is shown in Figure 18.16.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 18.16 Compatibility matrix.

Telegram: @uni_k

www.konkur.in

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction T_i that attempts to lock a node Q must follow these rules:

- Transaction T_i must observe the lock-compatibility function of Figure 18.16.
- Transaction T_i must lock the root of the tree first and can lock it in any mode.
- Transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode.
- Transaction T_i can lock a node Q in X, SIX, or IX mode only if T_i currently has the parent of Q locked in either IX or SIX mode.
- Transaction T_i can lock a node only if T_i has not previously unlocked any node (i.e., T_i is two phase).
- Transaction T_i can unlock a node Q only if T_i currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order. Deadlock is possible in the multiple-granularity protocol, as it is in the two-phase locking protocol.

As an illustration of the protocol, consider the tree of Figure 18.15 and these transactions:

- Suppose that transaction T_{21} reads record r_{a_2} in file F_a. Then, T_{21} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.
- Suppose that transaction T_{22} modifies record r_{a_9} in file F_a. Then, T_{22} needs to lock the database, area A_1, and file F_a (and in that order) in IX mode, and finally to lock r_{a_9} in X mode.
- Suppose that transaction T_{23} reads all the records in file F_a. Then, T_{23} needs to lock the database and area A_1 (and in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{24} reads the entire database. It can do so after locking the database in S mode.

We note that transactions T_{21}, T_{23}, and T_{24} can access the database concurrently. Transaction T_{22} can execute concurrently with T_{21}, but not with either T_{23} or T_{24}.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of:

- Short transactions that access only a few data items.
- Long transactions that produce reports from an entire file or set of files.

Telegram: @uni_k

www.konkur.in

The number of locks that an SQL query may need to acquire can usually be estimated based on the relation scan operations performed by a query. A relation scan, for example, would acquire a lock at a relation level, while an index scan that is expected to fetch only a few records may acquire an intention lock at the relation level and regular locks at the tuple level. In case the a transaction acquires a large number of tuple locks, the lock table may become overfull. To deal with this situation, the lock manager may perform **lock escalation**, replacing many lower level locks by a single higher level lock; in our example, a single relation lock could replace a large number of tuple locks.

18.4 Insert Operations, Delete Operations, and Predicate Reads

Until now, we have restricted our attention to read and write operations. This restriction limits transactions to data items already in the database. Some transactions require not only access to existing data items, but also the ability to create new data items. Others require the ability to delete data items. To examine how such transactions affect concurrency control, we introduce these additional operations:

- **delete**(Q) deletes data item Q from the database.
- **insert**(Q) inserts a new data item Q into the database and assigns Q an initial value.

An attempt by a transaction T_i to perform a read(Q) operation after Q has been deleted results in a logical error in T_i. Likewise, an attempt by a transaction T_i to perform a read(Q) operation before Q has been inserted results in a logical error in T_i. It is also a logical error to attempt to delete a nonexistent data item.

18.4.1 Deletion

To understand how the presence of **delete** instructions affects concurrency control, we must decide when a **delete** instruction conflicts with another instruction. Let I_i and I_j be instructions of T_i and T_j, respectively, that appear in schedule S in consecutive order. Let I_i = **delete**(Q). We consider several instructions I_j.

- I_j = read(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the read operation successfully.
- I_j = write(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the write operation successfully.
- I_j = **delete**(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_i will have a logical error.
- I_j = **insert**(Q). I_i and I_j conflict. Suppose that data item Q did not exist prior to the execution of I_i and I_j. Then, if I_i comes before I_j, a logical error results for T_i.

Telegram: @uni_k

www.konkur.in

If I_j comes before I_i, then no logical error results. Likewise, if Q existed prior to the execution of I_i and I_j, then a logical error results if I_j comes before I_i, but not otherwise.

We can conclude the following:

- Under the two-phase locking protocol, an exclusive lock is required on a data item before that item can be deleted.
- Under the timestamp-ordering protocol, a test similar to that for a write must be performed. Suppose that transaction T_i issues **delete**(Q).
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i was to delete has already been read by a transaction T_j with TS(T_j) > TS(T_i). Hence, the **delete** operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then a transaction T_j with TS(T_j) > TS(T_i) has written Q. Hence, this **delete** operation is rejected, and T_i is rolled back.
  - Otherwise, the **delete** is executed.

18.4.2 Insertion

We have already seen that an **insert**(Q) operation conflicts with a **delete**(Q) operation. Similarly, **insert**(Q) conflicts with a read(Q) operation or a write(Q) operation; no read or write can be performed on a data item before it exists.

Since an **insert**(Q) assigns a value to data item Q, an **insert** is treated similarly to a write for concurrency-control purposes:

- Under the two-phase locking protocol, if T_i performs an **insert**(Q) operation, T_i is given an exclusive lock on the newly created data item Q.
- Under the timestamp-ordering protocol, if T_i performs an **insert**(Q) operation, the values R-timestamp(Q) and W-timestamp(Q) are set to TS(T_i).

18.4.3 Predicate Reads and The Phantom Phenomenon

Consider transaction T_{30} that executes the following SQL query on the university database:

```
select count(*)
from instructor
where dept_name = 'Physics' ;
```

Transaction T_{30} requires access to all tuples of the *instructor* relation pertaining to the Physics department.

Telegram: @uni_k

www.konkur.in

Let T_{31} be a transaction that executes the following SQL insertion:

```
insert into instructor
        values (11111, 'Feynman', 'Physics', 94000);
```

Let S be a schedule involving T_{30} and T_{31}. We expect there to be potential for a conflict for the following reasons:

- If T_{30} uses the tuple newly inserted by T_{31} in computing **count**(*), then T_{30} reads a value written by T_{31}. Thus, in a serial schedule equivalent to S, T_{31} must come before T_{30}.
- If T_{30} does not use the tuple newly inserted by T_{31} in computing **count**(*), then in a serial schedule equivalent to S, T_{30} must come before T_{31}.

The second of these two cases is curious. T_{30} and T_{31} do not access any tuple in common, yet they conflict with each other! In effect, T_{30} and T_{31} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. As a result, the system could fail to prevent a nonserializable schedule. This problem is an instance of the **phantom phenomenon**.

Phantom phenomena can occur not just with inserts, but also with updates. Consider the situation we saw in Section 17.10, where a transaction T_i used an index to find only tuples with *dept_name* = "Physics", and as a result did not read any tuples with other department names. If another transaction T_j updates one of these tuples, changing its department name to Physics, a problem similar to the above problem occurs: even though T_i and T_j have not accessed any tuples in common, they do conflict with each other. This problem too is an instance of the phantom phenomenon. In general, the phantom phenomenon is rooted in predicate reads that conflict with inserts or updates that result in new/updated tuples that satisfy the predicate.

We can prevent these problems by allowing transaction T_{30} to prevent other transactions from creating new tuples in the *instructor* relation with *dept_name* = "Physics", and from updating the department name of an existing *instructor* tuple to Physics.

To find all *instructor* tuples with *dept_name* = "Physics", T_{30} must search either the whole *instructor* relation, or at least an index on the relation. Up to now, we have assumed implicitly that the only data items accessed by a transaction are tuples. However, T_{30} is an example of a transaction that reads information about what tuples are in a relation, and T_{31} is an example of a transaction that updates that information.

Clearly, it is not sufficient merely to lock the tuples that are accessed; the information used to find the tuples that are accessed by the transaction must also be locked.

Locking of information used to find tuples can be implemented by associating a data item with the relation; the data item represents the information used to find the tuples in the relation. Transactions, such as T_{30}, that read the information about what tuples are in a relation would then have to lock the data item corresponding to the

Telegram: @uni_k

www.konkur.in

relation in shared mode. Transactions, such as T_{31}, that update the information about what tuples are in a relation would have to lock the data item in exclusive mode. Thus, T_{30} and T_{31} would conflict on a real data item, rather than on a phantom. Similarly, transactions that use an index to retrieve tuples must lock the index itself.

Do not confuse the locking of an entire relation, as in multiple-granularity locking, with the locking of the data item corresponding to the relation. By locking the data item, a transaction only prevents other transactions from updating information about what tuples are in the relation. Locking is still required on tuples. A transaction that directly accesses a tuple can be granted a lock on the tuples even when another transaction has an exclusive lock on the data item corresponding to the relation itself.

The major disadvantage of locking a data item corresponding to the relation, or locking an entire index, is the low degree of concurrency— two transactions that insert different tuples into a relation are prevented from executing concurrently.

A better solution is an **index-locking** technique that avoids locking the whole index. Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. We eliminate the phantom phenomenon by imposing a locking protocol for indices. For simplicity we shall consider only B^+-tree indices.

As we saw in Chapter 14, every search-key value is associated with an index leaf node. A query will usually use one or more indices to access a relation. An insert must insert the new tuple in all indices on the relation. In our example, we assume that there is an index on *instructor* for attribute *dept_name*. Then, T_{31} must modify the leaf containing the key "Physics". If T_{30} reads the same leaf node to locate all tuples pertaining to the Physics department, then T_{30} and T_{31} conflict on that leaf node.

The **index-locking protocol** takes advantage of the availability of indices on a relation, by turning instances of the phantom phenomenon into conflicts on locks on index leaf nodes. The protocol operates as follows:

- Every relation must have at least one index.
- A transaction T_i can access tuples of a relation only after first finding them through one or more of the indices on the relation. For the purpose of the index-locking protocol, a relation scan is treated as a scan through all the leaves of one of the indices.
- A transaction T_i that performs a lookup (whether a range lookup or a point lookup) must acquire a shared lock on all the index leaf nodes that it accesses.
- A transaction T_i may not insert, delete, or update a tuple t_i in a relation r without updating all indices on r. The transaction must obtain exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update. For insertion and deletion, the leaf nodes affected are those that contain (after insertion) or contained (before deletion) the search-key value of the tuple. For updates, the leaf nodes affected are those that (before the modification) contained the old value of the search key, and nodes that (after the modification) contain the new value of the search key.

Telegram: @uni_k

www.konkur.in

- Locks are obtained on tuples as usual.
- The rules of the two-phase locking protocol must be observed.

Note that the index-locking protocol does not address concurrency control on internal nodes of an index; techniques for concurrency control on indices, which minimize lock conflicts, are presented in Section 18.10.2.

Locking an index leaf node prevents any update to the node, even if the update did not actually conflict with the predicate. A variant called key-value locking, which minimizes such false lock conflicts, is presented in Section 18.10.2 as part of index concurrency control.

As noted in Section 17.10, it would appear that the existence of a conflict between transactions depends on a low-level query-processing decision by the system that is unrelated to a user-level view of the meaning of the two transactions. An alternative approach to concurrency control acquires shared locks on predicates in a query, such as the predicate "*salary* > 90000" on the *instructor* relation. Inserts and deletes of the relation must then be checked to see if they satisfy the predicate; if they do, there is a lock conflict, forcing the insert or delete to wait till the predicate lock is released. For updates, both the initial value and the final value of the tuple must be checked against the predicate. Such conflicting inserts, deletes, and updates affect the set of tuples selected by the predicate, and they cannot be allowed to execute concurrently with the query that acquired the (shared) predicate lock. We call this protocol **predicate locking**;[1] predicate locking is not used in practice since it is more expensive to implement than the index-locking protocol and does not give significant additional benefits.

18.5 Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

18.5.1 Timestamps

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the database system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j). There are two simple methods for implementing this scheme:

[1]The term *predicate locking* was used for a version of the protocol that used shared and exclusive locks on predicates, and was thus more complicated. The version we present here, with only shared locks on predicates, is also referred to as **precision locking**.

Telegram: @uni_k

www.konkur.in

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if $\text{TS}(T_i) < \text{TS}(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

1. **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.
2. **R-timestamp**(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

18.5.2 The Timestamp-Ordering Protocol

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction T_i issues read(Q).
  - If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
  - If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and $\text{TS}(T_i)$.
- Suppose that transaction T_i issues write(Q).
  - If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
  - If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back.
  - Otherwise, the system executes the write operation and sets W-timestamp(Q) to $\text{TS}(T_i)$.

Telegram: @uni_k

www.konkur.in

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions T_{25} and T_{26}. Transaction T_{25} displays the contents of accounts A and B:

T_{25}: read(B);
read(A);
display($A + B$).

Transaction T_{26} transfers \$50 from account B to account A, and then displays the contents of both:

T_{26}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 18.17, $\text{TS}(T_{25}) < \text{TS}(T_{26})$, and the schedule is possible under the timestamp protocol.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa (see Exercise 18.27).

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.

The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If a transaction is suffering from repeated restarts, conflicting transactions need to be temporarily blocked to enable the transaction to finish.

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.

Telegram: @uni_k

www.konkur.in

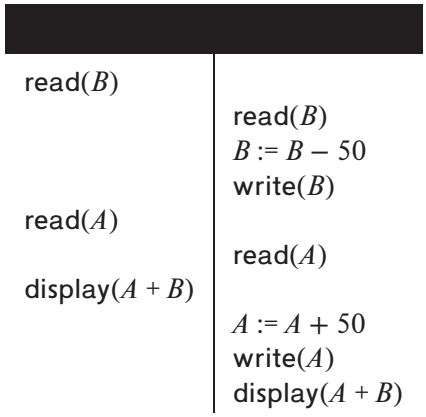


Figure 18.17 Schedule 3.

- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits (see Exercise 18.28).
- Recoverability alone can be ensured by tracking uncommitted writes and allowing a transaction T_i to commit only after the commit of any transaction that wrote a value that T_i read. Commit dependencies, outlined in Section 18.1.5, can be used for this purpose.

If the timestamp-ordering protocol is applied only to tuples, the protocol would be vulnerable to the phantom problems that we saw in Section 17.10 and Section 18.4.3.

To avoid this problem, the timestamp-ordering protocol could be applied to all data that is read by a transaction, including relation metadata and index data. In the context of locking-based concurrency control, the index-locking protocol, described in Section 18.4.3, is a more efficient alternative for avoiding the phantom problem; recall that the index-locking protocol obtains locks on index nodes, in addition to obtaining locks on tuples. The timestamp-ordering protocol can be similarly modified to treat each index node as a data item, with associated read and write timestamps, and to apply the timestamp-ordering tests on these data items, too. This extended version of the timestamp-ordering protocol avoids phantom problems and ensures serializability even with predicate reads.

18.5.3 Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol of Section 18.5.2. Let us consider schedule 4 of Figure 18.18 and apply the timestamp-ordering protocol. Since T_{27} starts before T_{28}, we shall assume that $\text{TS}(T_{27}) < \text{TS}(T_{28})$. The read($Q$) operation of T_{27} succeeds, as does the write(Q) operation of T_{28}. When T_{27} attempts its write(Q) operation,

Telegram: @uni_k

www.konkur.in

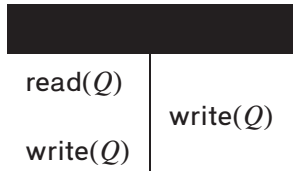


Figure 18.18 Schedule 4.

we find that TS(T_{27}) < W-timestamp(Q), since W-timestamp(Q) = TS(T_{28}). Thus, the write(Q) by T_{27} is rejected and transaction T_{27} must be rolled back.

Although the rollback of T_{27} is required by the timestamp-ordering protocol, it is unnecessary. Since T_{28} has already written Q, the value that T_{27} is attempting to write is one that will never need to be read. Any transaction T_i with TS(T_i) < TS(T_{28}) that attempts a read(Q) will be rolled back, since TS(T_i) < W-timestamp(Q). Any transaction T_j with TS(T_j) > TS(T_{28}) must read the value of Q written by T_{28}, rather than the value that T_{27} is attempting to write.

This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol of Section 18.5.2.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction T_i issues write(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

The difference between these rules and those of Section 18.5.2 lies in the second rule. The timestamp-ordering protocol requires that T_i be rolled back if T_i issues write(Q) and TS(T_i) < W-timestamp(Q). However, here, in those cases where TS(T_i) ≥ R-timestamp(Q), we ignore the obsolete write.

By ignoring the write, Thomas' write rule allows schedules that are not conflict serializable but are nevertheless correct. Those non-conflict-serializable schedules allowed satisfy the definition of *view serializable* schedules (see Note 18.1 on page 867). Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other

Telegram: @uni_k

www.konkur.in

protocols presented in this chapter. For example, schedule 4 of Figure 18.18 is not conflict serializable and, thus, is not possible under the two-phase locking protocol, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T_{27} would be ignored. The result is a schedule that is *view equivalent* to the serial schedule $<T_{27}, T_{28}>$.

18.6 Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for *monitoring* the system.

The **validation protocol** requires that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

1. **Read phase**. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. The validation test (described below) is applied to transaction T_i. This determines whether T_i is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. **Write phase**. If the validation test succeeds for transaction T_i, the temporary local variables that hold the results of any write operations performed by T_i are copied to the database. Read-only transactions omit this phase.

Each transaction must go through the phases in the order shown. However, phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction T_i:

1. ***StartTS***(T_i), the time when T_i started its execution.
2. ***ValidationTS***(T_i), the time when T_i finished its read phase and started its validation phase.
3. ***FinishTS***(T_i), the time when T_i finished its write phase.

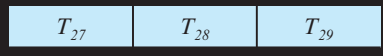
Telegram: @uni_k

www.konkur.in

1.

2.

3.



Telegram: @uni_k

www.konkur.in



We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp $ValidationTS(T_i)$. Thus, the value $TS(T_i) = ValidationTS(T_i)$ and, if $TS(T_j) < TS(T_k)$, then any produced schedule must be equivalent to a serial schedule in which transaction T_j appears before transaction T_k.

The **validation test** for transaction T_i requires that, for all transactions T_k with $TS(T_k) < TS(T_i)$, one of the following two conditions must hold:

1. $FinishTS(T_k) < StartTS(T_i)$. Since T_k completes its execution before T_i started, the serializability order is indeed maintained.
2. The set of data items written by T_k does not intersect with the set of data items read by T_i, and T_k completes its write phase before T_i starts its validation phase ($StartTS(T_i) < FinishTS(T_k) < ValidationTS(T_i)$). This condition ensures that the writes of T_k and T_i do not overlap. Since the writes of T_k do not affect the read of T_i, and since T_i cannot affect the read of T_k, the serializability order is indeed maintained.

As an illustration, consider again transactions T_{25} and T_{26}. Suppose that $TS(T_{25}) < TS(T_{26})$. Then, the validation phase succeeds in the schedule 6 in Figure 18.19. Note that the writes to the actual variables are performed only after the validation phase of T_{26}. Thus, T_{25} reads the old values of B and A, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked to enable the long transaction to finish.

Note also that the validation conditions result in a transaction T only being validated again the set of transactions T_i that finished after T started, and, further, are serialized before T. Transactions that finished before T started can be ignored in the validation tests. Transactions T_i that are serialized after T (that is, they have $ValidationTS(T_i) > ValidationTS(T)$) can also be ignored; when such a transaction T_i is validated, it would be validated against T if T finished after T started.

Telegram: @uni_k

www.konkur.in

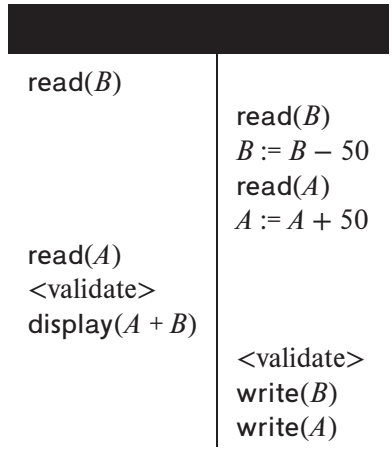


Figure 18.19 Schedule 6, a schedule produced by using validation.

This validation scheme is called the **optimistic concurrency-control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

It is possible to use $\text{TS}(T_i) = \textit{StartTS}(T_i)$ instead of $\textit{ValidationTS}(T_i)$ without affecting serializability. However, doing so may result in a transaction T_i entering the validation phase before a transaction T_j that has $\text{TS}(T_j) < \text{TS}(T_i)$. Then, the validation of T_i would have to wait for T_j to complete, so its read and write sets are completely known. Using *ValidationTS* avoids this problem.

18.7 Multiversion Schemes

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For example, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In **multiversion concurrency-control** schemes, each write(Q) operation creates a new **version** of Q. When a transaction issues a read(Q) operation, the concurrency-control manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

Telegram: @uni_k

www.konkur.in

18.7.1 Multiversion Timestamp Ordering

The timestamp-ordering protocol can be extended to a multiversion protocol. With each transaction T_i in the system, we associate a unique static timestamp, denoted by TS(T_i). The database system assigns this timestamp before the transaction starts execution, as described in Section 18.5.

With each data item Q, a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$ is associated. Each version Q_k contains three data fields:

1. **Content** is the value of version Q_k.
2. **W-timestamp**(Q_k) is the timestamp of the transaction that created version Q_k.
3. **R-timestamp**(Q_k) is the largest timestamp of any transaction that successfully read version Q_k.

A transaction—say, T_i—creates a new version Q_k of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by T_i. The system initializes the W-timestamp and R-timestamp to TS(T_i). It updates the R-timestamp value of Q_k whenever a transaction T_j reads the content of Q_k and R-timestamp(Q_k) < TS(T_j).

The **multiversion timestamp-ordering scheme** presented next ensures serializability. The scheme operates as follows: Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.
2. If transaction T_i issues write(Q), and if TS(T_i) < R-timestamp(Q_k), then the system rolls back transaction T_i. On the other hand, if TS(T_i) = W-timestamp(Q_k), the system overwrites the contents of Q_k; otherwise (if TS(T_i) > R-timestamp(Q_k)), it creates a new version of Q.

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is "too late" in doing a write. More precisely, if T_i attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

The **valid interval** of a version Q_i of Q with W-timestamp t is defined as follows: if Q_i is the latest version of Q, the interval is $[t, \infty]$; otherwise let the next version of Q have timestamp s; then the valid interval is $[t, s)$. You can easily verify that reads by a transaction with timestamp t_i return the content of the version whose valid interval contains t_i.

Versions that are no longer needed are removed according to the following rule: Suppose that there are two versions, Q_k and Q_j, of a data item, and that both versions

Telegram: @uni_k

www.konkur.in

have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again, and can be deleted.

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait. In typical database systems, where reading is a more frequent operation than is writing, this advantage may be of major practical significance.

The scheme, however, suffers from two undesirable properties. First, the reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one. Second, the conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive. Section 18.7.2 describes an algorithm to alleviate this problem.

This multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness. It can be extended in the same manner as the basic timestamp-ordering scheme to make it recoverable and cascadeless.

18.7.2 Multiversion Two-Phase Locking

The **multiversion two-phase locking protocol** attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between **read-only transactions** and **update transactions**.

Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.

The database system assigns read-only transactions a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads. Thus, when a read-only transaction T_i issues a read(Q), the value returned is the contents of the version whose timestamp is the largest timestamp less than or equal to TS(T_i).

When an update transaction reads an item, it gets a shared lock on the item and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value ∞, a value greater than that of any possible timestamp.

When the update transaction T_i completes its actions, it carries out commit processing; only one update transaction is allowed to perform commit processing at a time. First, T_i sets the timestamp on every version it has created to 1 more than the value of ts-counter; then, T_i increments ts-counter by 1, and commits.

Read-only transactions see the old value of ts-counter until T_i has successfully committed. As a result, read-only transactions that start after T_i commits will see the values updated by T_i, whereas those that start before T_i commits will see the value before the updates by T_i. In either case, read-only transactions never need to wait for

Telegram: @uni_k

www.konkur.in



locks. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Versions are deleted in a manner like that of multiversion timestamp ordering. Suppose there are two versions, Q_k and Q_j, of a data item, and that both versions have a timestamp less than or equal to the timestamp of the oldest read-only transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again and it can be deleted.

18.8 Snapshot Isolation

Snapshot isolation is a particular type of concurrency-control scheme that has gained wide acceptance in commercial and op

Telegram: @uni_k

www.konkur.in

PostgreSQL, and SQL Server. We introduced snapshot isolation in Section 17.9.3. Here, we take a more detailed look into how it works.

Conceptually, snapshot isolation involves giving a transaction a "snapshot" of the database at the time when it begins its execution. It then operates on that snapshot in complete isolation from concurrent transactions. The data values in the snapshot consist only of values written by committed transactions. This isolation is ideal for read-only transactions since they never wait and are never aborted by the concurrency manager.

Transactions that update the database potentially have conflicts with other transactions that update the database. Updates performed by a transaction must be validated before the transaction is allowed to commit. We describe how validation is performed, later in this section. Updates are kept in the transaction's private workspace until the transaction is validated, at which point the updates are written to the database.

When a transaction T is allowed to commit, the transition of T to the committed state and the writing of all of the updates made by T to the database must be conceptually done as an atomic action so that any snapshot created for another transaction either includes all updates by transaction T or none of them.

18.8.1 Multiversioning in Snapshot Isolation

To implement snapshot isolation, transactions are given two timestamps. The first timestamp, $StartTS(T_i)$, is the time at which transaction T_i started. The second timestamp, $CommitTS(T_i)$ is the time when the transaction T_i requested validation.

Note that timestamps can be wall clock time, as long as no two transactions are given the same timestamp, but they are usually assigned from a counter that is incremented every time a transaction enters its validation phase.

Snapshot isolation is based on multiversioning, and each transaction that updates a data item creates a version of the data item. Versions have only one timestamp, which is the write timestamp, indicating when the version was created. The timestamp of a version created by transaction T_i is set to $CommitTS(T_i)$. (Since updates to the database are also only made after validation of the transaction T_i, $CommitTS(T_i)$ is available when a version is created.)[2]

When a transaction T_i reads a data item, the latest version of the data item whose timestamp is $\leq StartTS(T_i)$ is returned to T_i. Thus, T_i does not see the updates of any transactions that committed after T_i started, while it does see the updates of all transactions that commit before it started. As a result, T_i effectively sees a snapshot of the database as of the time it started.[3]

[2]Many implementations create versions even before the transaction starts validation; since the version timestamp is not available at this point, the timestamp is set to infinity initially, and is updated to the correct value at the time of validation. Further optimizations are used in actual implementations, but we ignore them for simplicity.

[3]To efficiently find the correct version of a data item for a given timestamp, many implementations store not only the timestamp when a version was created, but also the timestamp when the next version was created, which can be considered an *invalidation timestamp* for that version; the version is valid between the creation and invalidation timestamps. The current version of a data item has the

Telegram: @uni_k

www.konkur.in

18.8.2 Validation Steps for Update Transactions

Deciding whether or not to allow an update transaction to commit requires some care. Potentially, two transactions running concurrently might both update the same data item. Since these two transactions operate in isolation using their own private snapshots, neither transaction sees the update made by the other. If both transactions are allowed to write to the database, the first update written will be overwritten by the second. The result is a **lost update**. This must be prevented. There are two variants of snapshot isolation, both of which prevent lost updates. They are called *first committer wins* and *first updater wins*. Both approaches are based on testing the transaction against concurrent transactions.

A transaction T_j is said to be **concurrent with** a given transaction T_i if it was active or partially committed at any point from the start of T up to the point when validation of T_i started. Formally, T_j is concurrent with T_i if either

$$StartTS(T_j) \leq StartTS(T_i) \leq CommitTS(T_j), \text{ or}$$
$$StartTS(T_i) \leq StartTS(T_j) \leq CommitTS(T_i).$$

Under **first committer wins**, when a transaction T_i starts validation, the following actions are performed as part of validation, after its *CommitTS* is assigned. (We assume for simplicity that only one transaction performs validation at a time, although real implementations do support concurrent validation.)

- A test is made to see if any transaction that was concurrent with T has already written an update to the database for some data item that T intends to write. This can be done by checking for each data item d that T_i intends to write, whether there is a version of the data item d whose timestamp is between $StartTS(T_i)$ and $CommitTS(T_i)$.[4]
- If any such data item is found, then T_i aborts.
- If no such data item is found, then T commits and its updates are written to the database.

This approach is called "first committer wins" because if transactions conflict, the first one to be tested using the above rule succeeds in writing its updates, while the subsequent ones are forced to abort. Details of how to implement these tests are addressed in Exercise 18.15.

Under **first updater wins**, the system uses a locking mechanism that applies only to updates (reads are unaffected by this, since they do not obtain locks). When a transaction T_i attempts to update a data item, it requests a *write lock* on that data item. If the lock is not held by a concurrent transaction, the following steps are taken after the lock is acquired:

- If the item has been updated by any concurrent transaction, then T_i aborts.

[4]There are alternative implementations, based on keeping track of read and

Telegram: @uni_k

www.konkur.in

- Otherwise T_i may proceed with its execution, including possibly committing.

If, however, some other concurrent transaction T_j already holds a write lock on that data item, then T_i cannot proceed, and the following rules are followed:

- T_i waits until T_j aborts or commits.
  - If T_j aborts, then the lock is released and T_i can obtain the lock. After the lock is acquired, the check for an update by a concurrent transaction is performed as described earlier: T_i aborts if a concurrent transaction had updated the data item, and it proceeds with its execution otherwise.
  - If T_j commits, then T_i must abort.

Locks are released when the transaction commits or aborts.

This approach is called "first updater wins" because if transactions conflict, the first one to obtain the lock is the one that is permitted to commit and perform its update. Those that attempt the update later abort unless the first updater subsequently aborts for some other reason. (As an alternative to waiting to see if the first updater T_j aborts, a subsequent updater T_i can be aborted as soon as it finds that the write lock it wishes to obtain is held by T_j.)

18.8.3 Serializability Issues and Solutions

Snapshot isolation is attractive in practice because transactions that read a lot of data (typically for data analysis) do not interfere with shorter update transactions (typically used for transaction processing). With two-phase locking, such long read-only transactions would block update transactions for long periods of time, which is often unacceptable.

It is worth noting that integrity constraints that are enforced by the database, such as primary-key and foreign-key constraints, cannot be checked on a snapshot; otherwise it would be possible for two concurrent transactions to insert two tuples with the same primary key value, or for a transaction to insert a foreign key value that is concurrently deleted from the referenced table. This problem is handled by checking these constraints on the current state of the database, rather than on the snapshot, as part of validation at the time of commit.

Even with the above fix, there is still a serious problem with the snapshot isolation scheme as we have presented it and as it is implemented in practice: *snapshot isolation does* **not** *ensure serializability!*

Next we give examples of possible nonserializable executions under snapshot isolation. We then outline the *serializable snapshot isolation* technique that is supported by some databases, which extends the snapshot isolation technique to ensure serializability. Snapshot isolation implementations that do not support serializable snapshot isolation often support SQL extensions that allow the programmer to ensure serializability even with snapshot isolation; we study these extensions at the end of the section.

Telegram: @uni_k

www.konkur.in

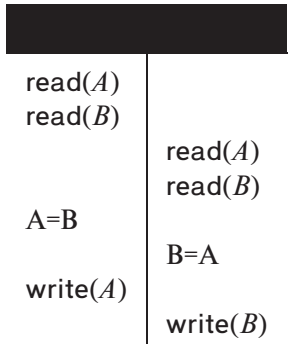


Figure 18.20 Nonserializable schedule under snapshot isolation.

- Consider the transaction schedule shown in Figure 18.20. Two concurrent transactions T_i and T_j both read data items A and B. T_i sets $A = B$ and writes A, while T_j sets $B = A$ and writes B. Since T_i and T_j are concurrent, under snapshot isolation neither transaction sees the update by the other in its snapshot. But, since they update different data items, both are allowed to commit regardless of whether the system uses the first-update-wins policy or the first-committer-wins policy.

  However, the execution is not serializable, since it results in swapping of the values of A and B, whereas any serializable schedule would set both A and B to the same value: either the initial value of A or the initial value of B, depending on the order of T_i and T_j.

  It can be easily seen that the precedence graph has a cycle. There is an edge in the precedence graph from T_i to T_j because T_i reads the value of A that existed before T_j writes A. There is also an edge in the precedence graph from T_j to T_i because T_j reads the value of B that existed before T_i writes B. Since there is a cycle in the precedence graph, the result is a nonserializable schedule.

  This situation, where each of a pair of transactions has read a data item that is written by the other, but the set of data items written by the two transactions do not have any data item in common, is referred to as **write skew**.

- As another example of write skew, consider a banking scenario. Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200, respectively. Suppose that transaction T_{36} withdraws \$200 from the checking account, after verifying the integrity constraint by reading both balances. Suppose that concurrently transaction T_{37} withdraws \$200 from the savings account, again after verifying the integrity constraint. Since each of the transactions checks the integrity constraint on its own snapshot, if they run concurrently each will believe that the sum of the balances after the withdrawal is \$100, and therefore its withdrawal does not violate

Telegram: @uni_k

www.konkur.in

the constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both of them can commit.

Unfortunately, in the final state after both T_{36} and T_{37} have committed, the sum of the balances is \$100, violating the integrity constraint. Such a violation could never have occurred in any serial execution of T_{36} and T_{37}.

- Many financial applications create consecutive sequence numbers, for example to number bills, by taking the maximum current bill number and adding 1 to the value to get a new bill number. If two such transactions run concurrently, each would see the same set of bills in its snapshot, and each would create a new bill with the same number. Both transactions pass the validation tests for snapshot isolation, since they do not update any tuple in common. However, the execution is not serializable; the resultant database state cannot be obtained by any serial execution of the two transactions. Creating two bills with the same number could have serious legal implications.

  The above problem is in fact an example of the phantom phenomenon, which we saw in Section 18.4.3, since the insert performed by each transaction conflicts with the read performed by the other transaction to find the maximum bill number, but the conflict is not detected by snapshot isolation.[5]

The problems listed above seem to indicate that the snapshot isolation technique is vulnerable to many serializability problems and should never be used. However, serializability problems are relatively rare for two reasons:

1. The fact that the database must check integrity constraints at the time of commit, and not on a snapshot, helps avoid inconsistencies in many situations. For example, in the financial application example that we saw earlier, the bill number would likely have been declared as a primary key. The database system would detect the primary key violation outside the snapshot and roll back one of the two transactions.

   It was shown that primary key constraints ensured that all transactions in a popular transaction processing benchmark, TPC-C, were free from nonserializability problems, when executed under snapshot isolation. This was viewed as an indication that such problems are rare. However, they do occur occasionally, and when they occur they must be dealt with.[6]

2. In many applications that are vulnerable to serializability problems, such as skew writes, on some data items, the transactions conflict on other data items, ensuring

[5] The SQL standard uses the term *phantom problem* to refer to nonrepeatable predicate reads, leading some to claim that snapshot isolation avoids the phantom problem; however, such a claim is not valid under our definition of phantom conflict.

[6] For example, the problem of duplicate bill numbers actually occurred several times in a financial application in I.I.T. Bombay, where (for reasons too complex to discuss here) the bill number was not a primary key, and it was detected by financial auditors.

Telegram: @uni_k

www.konkur.in

such transactions cannot execute concurrently; as a result, the execution of such transactions under snapshot isolation remains serializable.

Nonserializable may nevertheless occur with snapshot isolation. The impact of nonserializable execution due to snapshot isolation is not very severe for many applications. For example, consider a university application that implements enrollment limits for a course by counting the current enrollment before allowing registration. Snapshot isolation could allow the class enrollment limit to be exceeded. However, this may happen very rarely, and if it does, having one extra student in a class is usually not a major problem. The fact that snapshot isolation allows long read transactions to execute without blocking updaters is a large enough benefit for many such applications to live with occasional glitches.

Nonserializability may not be acceptable for many other applications, such as financial applications. There are several possible solutions.

- A modified form of snapshot isolation, called serializable snapshot isolation, can be used if it is supported by the database system. This technique extends the snapshot isolation technique in a way that ensures serializability.
- Some systems allow different transactions to run under different isolation levels, which can be used to avoid the serializability problems mentioned above.
- Some systems that support snapshot isolation provide a way for SQL programmers to create artificial conflicts, using a **for update** clause in SQL, which can be used to ensure serializability.

We briefly outline each of these solutions below.

Since version 9.1, PostgreSQL implements a technique called serializable snapshot isolation, which ensures serializability; in addition, PostgreSQL versions from 9.1 onwards include an index-locking-based technique to provide protection against phantom problems.

The intuition behind the **serializable snapshot isolation** (**SSI**) protocol is as follows: Suppose we track all conflicts (i.e., write-write, read-write, and write-read conflicts) between transactions. Recall from Section 17.6 that we can construct a transaction precedence graph which has a directed edge from T_1 to T_2 if transactions T_1 and T_2 have conflicting operations on a tuple, with T_1's action preceding T_2's action. As we saw in Section 17.6, one way to ensure serializability is to look for cycles in the transaction precedence graph and roll back transactions if a cycle is found.

The key reason for loss of serializability with snapshot isolation is that read-write conflicts, where a transaction T_1 writes a version of an object, and a transaction T_2 subsequently reads an earlier version of the object, are not tracked by snapshot isolation. This conflict can be represented by a read-write conflict edge from T_2 to T_1.

It has been shown that in all cases where snapshot isolation allows nonserializable schedules, there must be a transaction that has both an incoming read-write conflict

Telegram: @uni_k

www.konkur.in

edge and an outgoing read-write conflict edge (all other cases of cycles in the conflict graph are caught by the snapshot isolation rules). Thus, serializable snapshot isolation implementations track all read-write conflicts between concurrent transactions to detect if a transaction has both an incoming and an outgoing read-write conflict edge. If such a situation is detected, one of the transactions involved in the read-write conflicts is rolled back. This check is significantly cheaper than tracking all conflicts and looking for cycles, although it may result in some unnecessary rollbacks.

It is also worth mentioning that the technique used by PostgreSQL to prevent phantoms uses index locking, but the locks are not held in a two-phase manner. Instead, they are used to detect potential conflicts between concurrent transactions and must be retained for some time even after a transaction commits, to allow checks against other concurrent transactions. The index-locking technique used by PostgreSQL also does not result in any deadlocks.

SQL Server offers the option of allowing some transactions to run under *snapshot* isolation, while allowing others to run under the *serializable* isolation level. Running long read-only transactions under the snapshot isolation level while running update transactions under the serializable isolation level ensures that the read-only transaction does not block updaters, while also ensuring that the above anomalies cannot occur.

In Oracle versions till at least Oracle 12c (to the best of our knowledge), and in PostgreSQL versions prior to 9.1, the *serializable* isolation level actually implements snapshot isolation. As a result, even with the isolation level set to serializable, it is possible that the database permits some schedules that are not serializable.

If an application has to run under snapshot isolation, on several of these databases an application developer can guard against certain snapshot anomalies by appending a **for update** clause to the SQL select query as illustrated below:

```
select *
from instructor
where ID = 22222
for update;
```

Adding the **for update** clause causes the system to treat data that are read as if they had been updated for purposes of concurrency control. In our first example of write skew shown in Figure 18.20, if the **for update** clause were appended to the select queries that read the values of A and B, only one of the two concurrent transactions would be allowed to commit since it appears that both transactions have updated both A and B.

Formal methods exist (see the online bibliographical notes) to determine whether a given mix of transactions runs the risk of nonserializable execution under snapshot isolation and to decide on what conflicts to introduce (using the **for update** clause, for example) to ensure serializability. Such methods can work only if we know in advance what transactions are being executed. In some applications, all transactions are from a predetermined set of transactions, making this analysis possible. However, if the application allows unrestricted, ad hoc transactions, then no such analysis is possible.

Telegram: @uni_k

www.konkur.in

18.9 Weak Levels of Consistency in Practice

In Section 17.8, we discussed the isolation levels specified by the SQL standard: serializable, repeatable read, read committed, and read uncommitted. In this section, we first briefly outline some older terminology relating to consistency levels weaker than serializability and relate it to the SQL standard levels. We then discuss the issue of concurrency control for transactions that involve user interaction, an issue that we briefly discussed in Section 17.8.

18.9.1 Degree-Two Consistency

The purpose of **degree-two consistency** is to avoid cascading aborts without necessarily ensuring serializability. The locking protocol for degree-two consistency uses the same two lock modes that we used for the two-phase locking protocol: shared (S) and exclusive (X). A transaction must hold the appropriate lock mode when it accesses a data item, but two-phase behavior is not required.

In contrast to the situation in two-phase locking, S-locks may be released at any time, and locks may be acquired at any time. Exclusive locks, however, cannot be released until the transaction either commits or aborts. Serializability is not ensured by this protocol. Indeed, a transaction may read the same data item twice and obtain different results. In Figure 18.21, T_{32} reads the value of Q before that value is written by T_{33}, and again after it is written by T_{33}.

Reads are not repeatable, but since exclusive locks are held until transaction commit, no transaction can read an uncommitted value. Thus, degree-two consistency is one particular implementation of the read-committed isolation level.

It is interesting to note that with degree-two consistency, a transaction that is scanning an index may potentially see two versions of a record that was updated while the scan was in progress and may also potentially see neither version! For example,

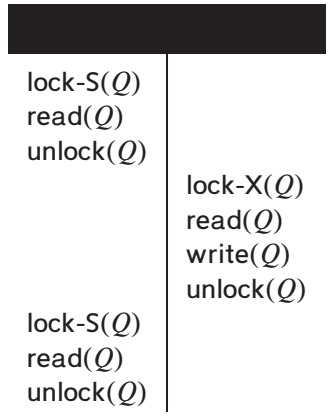


Figure 18.21 Nonserializable schedule with degree-two consistency.

Telegram: @uni_k

www.konkur.in

consider a relation $r(A, B, C)$, with primary key A, with an index on attribute B. Now consider a query that is scanning the relation r using the index on attribute B, using degree-two consistency. Suppose there is a concurrent update to a tuple $t_1 \in r$ that updates attribute $t_1.B$ from v_1 to v_2. Such an update requires deletion of an entry corresponding to value v_1 from the index and insertion of a new entry corresponding to v_2. Now, the scan of r could possibly scan the index node corresponding to v_1 after the old tuple is deleted there but visit the index node corresponding to v_2 before the updated tuple is inserted in that node. Then, the scan would completely miss the tuple, even though it should have seen either the old value or the new value of t_1. Further, a scan using degree-two consistency could possibly visit the node corresponding to v_1 before the delete, and the node corresponding to v_2 after the insert, and thereby see two versions of t_1, one from before the update and one from after the update. (This problem would not arise if the scan and the update both used two-phase locking.)

18.9.2 Cursor Stability

Cursor stability is a form of degree-two consistency designed for programs that iterate over tuples of a relation by using cursors. Instead of locking the entire relation, cursor stability ensures that:

- The tuple that is currently being processed by the iteration is locked in shared mode. Once the tuple is processed, the lock on the tuple can be released.
- Any modified tuples are locked in exclusive mode until the transaction commits.

These rules ensure that degree-two consistency is obtained. But locking is not done in a two-phase manner, and serializability is not guaranteed. Cursor stability is used in practice on heavily accessed relations as a means of increasing concurrency and improving system performance. Applications that use cursor stability must be coded in a way that ensures database consistency despite the possibility of nonserializable schedules. Thus, the use of cursor stability is limited to specialized situations with simple consistency constraints.

When supported by the database, snapshot isolation is a better alternative to degree-two consistency as well as cursor stability, since it offers a similar or even better level of concurrency while reducing the risk of nonserializable executions.

18.9.3 Concurrency Control Across User Interactions

Concurrency-control protocols usually consider transactions that do not involve user interaction. Consider the airline seat selection example from Section 17.8, which involved user interaction. Suppose we treat all the steps from when the seat availability is initially shown to the user, until the seat selection is confirmed, as a single transaction.

If two-phase locking is used, the entire set of seats on a flight would be locked in shared mode until the user has completed the seat selection, and no other transaction would be able to update the seat allocation information in this period. Such locking

Telegram: @uni_k

www.konkur.in

would be a very bad idea since a user may take a long time to make a selection, or even just abandon the transaction without explicitly cancelling it. Timestamp protocols or validation could be used instead, which avoid the problem of locking, but both these protocols would abort the transaction for a user *A* if any other user *B* has updated the seat allocation information, even if the seat selected by *B* does not conflict with the seat selected by user *A*. Snapshot isolation is a good option in this situation, since it would not abort the transaction of user *A* as long as *B* did not select the same seat as *A*.

However, snapshot isolation requires the database to remember information about updates performed by a transaction even after it has committed, as long as any other concurrent transaction is still active, which can be problematic for long-duration transactions.

Another option is to split a transaction that involves user interaction into two or more transactions, such that no transaction spans a user interaction. If our seat selection transaction is split thus, the first transaction would read the seat availability, while the second transaction would complete the allocation of the selected seat. If the second transaction is written carelessly, it could assign the selected seat to the user, without checking if the seat was meanwhile assigned to some other user, resulting in a lost-update problem. To avoid the problem, as we outlined in Section 17.8, the second transaction should perform the seat allocation only if the seat was not meanwhile assigned to some other user.

The above idea has been generalized in an alternative concurrency control scheme, which uses version numbers stored in tuples to avoid lost updates. The schema of each relation is altered by adding an extra *version_number* attribute, which is initialized to 0 when the tuple is created. When a transaction reads (for the first time) a tuple that it intends to update, it remembers the version number of that tuple. The read is performed as a stand-alone transaction on the database, and hence any locks that may be obtained are released immediately. Updates are done locally and copied to the database as part of commit processing, using the following steps which are executed atomically (i.e., as part of a single database transaction):

- For each updated tuple, the transaction checks if the current version number is the same as the version number of the tuple when it was first read by the transaction.
    1. If the version numbers match, the update is performed on the tuple in the database, and its version number is incremented by 1.
    2. If the version numbers do not match, the transaction is aborted, rolling back all the updates it performed.
- If the version number check succeeds for all updated tuples, the transaction commits. It is worth noting that a timestamp could be used instead of the version number without impacting the scheme in any way.

Telegram: @uni_k

www.konkur.in

Observe the close similarity between the preceding scheme and snapshot isolation. The version number check implements the first-committer-wins rule used in snapshot isolation, and it can be used even if the transaction was active for a very long time. However, unlike snapshot isolation, the reads performed by a transaction may not correspond to a snapshot of the database; and unlike the validation-based protocol, reads performed by the transaction are not validated.

We refer to the above scheme as **optimistic concurrency control without read validation**. Optimistic concurrency control without read validation provides a weak level of serializability, and it does not ensure serializability. A variant of this scheme uses version numbers to validate reads at the time of commit, in addition to validating writes, to ensure that the tuples read by the transaction were not updated subsequent to the initial read; this scheme is equivalent to the optimistic concurrency-control scheme which we saw earlier.

This scheme has been widely used by application developers to handle transactions that involve user interaction. An attractive feature of the scheme is that it can be implemented easily on top of a database system. The validation and update steps performed as part of commit processing are then executed as a single transaction in the database, using the concurrency-control scheme of the database to ensure atomicity for commit processing. The scheme is also used by the Hibernate object-relational mapping system (Section 9.6.2), and other object-relational mapping systems, where it is referred to as optimistic concurrency control (even though reads are not validated by default). Hibernate and other object-relational mapping systems therefore perform the version number checks transparently as part of commit processing. (Transactions that involve user interaction are called **conversations** in Hibernate to differentiate them from regular transactions; validation using version numbers is particularly useful for such transactions.)

Application developers must, however, be aware of the potential for non-serializable execution, and they must restrict their usage of the scheme to applications where non-serializability does not cause serious problems.

18.10 Advanced Topics in Concurrency Control

Instead of using two-phase locking, special-purpose concurrency control techniques can be used for index structures, resulting in improved concurrency. When using main-memory databases, conversely, index concurrency control can be simplified. Further, concurrency control actions often become bottlenecks in main-memory databases, and techniques such as latch-free data structures have been designed to reduce concurrency control overheads. Instead of detecting conflicts at the level of reads and writes, it is possible to consider operations, such as increment of a counter, as basic operations, and perform concurrency control on the basis of conflicts between operations. Certain applications require guarantees on transaction completion time. Specialized concurrency control techniques have been developed for such applications.

Telegram: @uni_k

www.konkur.in

18.10.1 Online Index Creation

When we are dealing with large volumes of data (ranging in the terabytes), operations such as creating an index can take a long time—perhaps hours or even days. When the operation finishes, the index contents must be consistent with the contents of the relation, and all further updates to the relation must maintain the index.

One way of ensuring that the data and the index are consistent is to block all updates to the relation while the index is created, for example by getting a shared lock on the relation. After the index is created, and the relation metadata are updated to reflect the existence of the index locks can be released. Subsequent update transactions will find the index, and carry out index maintenance as part of the transaction.

However, the above approach would make the system unavailable for updates to the relation for a very long time, which is unacceptable. Instead, most database systems support **online index creation**, which allows relation updates to occur even as the index is being created. Online index creation can be carried out as follows:

1. Index creation gets a snapshot of the relation and uses it to create the index; meanwhile, the system logs all updates to the relation that happen after the snapshot is created.
2. When the index on the snapshot data is complete, it is not yet ready for use, since subsequent updates are missing. At this point, the log of updates to the relation is used to update the index. But while the index update is being carried out, further updates may be happening on the relation.
3. The index update then obtains a shared lock on the relation to prevent further updates and applies all remaining updates to the index. At this point, the index is consistent with the contents of the relation. The relation metadata are then updated to indicate the existence of the new index. Subsequently all locks are released.
   Any transaction that executes after this will see the existence of the index; if the transaction updates the relation, it will also update the index.

Creation of materialized views that are maintained immediately, as part of the transaction that updates any of the relations used in the view, can also benefit from online construction techniques that are similar to online index construction. The query defining the view is executed on a snapshot of the participating relations, and subsequent updates are logged. The updates are applied to the materialized view, with a final phase of locking and catching up similar to the case of online index creation.

Schema changes such as adding or deleting attributes or constraints can also have a significant impact if relations are locked while the schema change is implemented on all tuples.

- For adding or deleting attributes, a version number can be kept with each tuple, and tuples can be updated in the background, or whenever they are accessed; the version number is used to determine if the schema change has already been applied

Telegram: @uni_k

www.konkur.in

to the tuple, and the schema change is applied to the tuple if it has not already been applied.

- Adding of constraints requires that existing data must be checked to ensure that the constraint is satisfied. For example, adding a primary or unique key constraint on an attribute ID requires checking of existing tuples to ensure that no two tuples have the same ID value. Online addition of such constraints is done in a manner similar to online index construction, by checking the constraints on a relation snapshot, while keeping a log of updates that occur after the snapshot. The updates in the log must then be checked to ensure that they do not violate the constraint. In a final catch-up phase, the constraint is checked on any remaining updates in the log and added to the relation metadata while holding a shared lock on the relation.

18.10.2 Concurrency in Index Structures

It is possible to treat access to index structures like any other database structure and to apply the concurrency-control techniques discussed earlier. However, since indices are accessed frequently, they would become a point of great lock contention, leading to a low degree of concurrency. Luckily, indices do not have to be treated like other database structures; it is desirable to release index locks early, in a non-two-phase manner, to maximize concurrency. In fact, it is perfectly acceptable for a transaction to perform a lookup on an index twice and to find that the structure of the index has changed in between, as long as the index lookup returns the correct set of tuples. Informally, it is acceptable to have nonserializable concurrent access to an index, as long as the accuracy of the index is maintained; we formalize this notion next.

Operation serializability for index operations is defined as follows: A concurrent execution of index operations on an index is said to be serializable if there is a serialization order of the operations that is consistent with the results that each index operation in the concurrent execution sees, as well as with the final state of the index after all the operations have been executed. Index concurrency control techniques must ensure that any concurrent execution of index operations is serializable.

We outline two techniques for managing concurrent access to B^+-trees as well as an index-concurrency control technique to prevent the phantom phenomenon. The online bibliographical notes reference other techniques for B^+-trees as well as techniques for other index structures. The techniques that we present for concurrency control on B^+-trees are based on locking, but neither two-phase locking nor the tree protocol is employed. The algorithms for lookup, insertion, and deletion are those used in Chapter 14, with only minor modifications.

The first technique is called the **crabbing protocol**:

- When searching for a key value, the crabbing protocol first locks the root node in shared mode. When traversing down the tree, it acquires a shared lock on the child node to be traversed further. After acquiring the lock on the child node, it releases the lock on the parent node. It repeats this process until it reaches a leaf node.

Telegram: @uni_k

www.konkur.in

- When inserting or deleting a key value, the crabbing protocol takes these actions:
  - It follows the same protocol as for searching until it reaches the desired leaf node. Up to this point, it obtains (and releases) only shared locks.
  - It locks the leaf node in exclusive mode and inserts or deletes the key value.
  - If it needs to split a node or coalesce it with its siblings, or redistribute key values between siblings, the crabbing protocol locks the parent of the node in exclusive mode. After performing these actions, it releases the locks on the node and siblings.

    If the parent requires splitting, coalescing, or redistribution of key values, the protocol retains the lock on the parent, and splitting, coalescing, or redistribution propagates further in the same manner. Otherwise, it releases the lock on the parent.

The protocol gets its name from the way in which crabs advance by moving sideways, moving the legs on one side, then the legs on the other, and so on alternately. The progress of locking while the protocol both goes down the tree and goes back up (in case of splits, coalescing, or redistribution) proceeds in a similar crab-like manner.

Once a particular operation releases a lock on a node, other operations can access that node. There is a possibility of deadlocks between search operations coming down the tree, and splits, coalescing, or redistribution propagating up the tree. The system can easily handle such deadlocks by restarting the search operation from the root, after releasing the locks held by the operation.

Locks that are held for a short duration, instead of being held in a two-phase manner, are often referred to as **latches**. Latches are used internally in databases to achieve mutual exclusion on shared data structures. In the above case, locks are held in a way that does not ensure mutual exclusion during an insert or delete operation, yet the resultant execution of index operations is serializable.

The second technique achieves even more concurrency, avoiding even holding the lock on one node while acquiring the lock on another node; thereby, deadlocks are avoided, and concurrency is increased. This technique uses a modified version of B^+-trees called **B-link trees**; B-link trees require that every node (including internal nodes, not just the leaves) maintain a pointer to its right sibling. This pointer is required because a lookup that occurs while a node is being split may have to search not only that node but also that node's right.

Unlike the crabbing protocol, the **B-link-tree locking protocol** holds locks on only one internal node at a time. The protocol releases the lock on the current internal node before requesting a lock on a child node (when traversing downwards), or on a parent node (while traversing upwards during a split or merge). Doing so can result in anomalies: for example, between the time the lock on a node is released and the lock on a parent is requested, a concurrent insert or delete on a sibling may cause a split or merge on the parent, and the original parent node may no longer be a parent of the

Telegram: @uni_k

www.konkur.in

child node when it is locked. The protocol detects and handles such situations, ensuring operation serializability while avoiding deadlocks between operations and increasing concurrency compared to the crabbing protocol.

The phantom phenomenon, where conflicts between a predicate read and an insert or update are not detected, can allow nonserializable executions to occur. The index-locking technique, which we saw in Section 18.4.3, prevents the phantom phenomenon by locking index leaf nodes in a two-phase manner. Instead of locking an entire index leaf node, some index concurrency-control schemes use **key-value locking** on individual key values, allowing other key values to be inserted or deleted from the same leaf. Key-value locking thus provides increased concurrency.

Using key-value locking naïvely, however, would allow the phantom phenomenon to occur; to prevent the phantom phenomenon, the **next-key locking** technique is used. In this technique, every index lookup must lock not only the keys found within the range (or the single key, in case of a point lookup) but also the next-key value—that is, the key value just greater than the last key value that was within the range. Also, every insert must lock not only the value that is inserted, but also the next-key value. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict on the key value next to the inserted key value. Similarly, deletes must also lock the next-key value to the value being deleted to ensure that conflicts with subsequent range lookups of other queries are detected.

18.10.3 Concurrency Control in Main-Memory Databases

With data stored on hard disk, the cost of I/O operations often dominates the cost of transaction processing. When disk I/O is the bottleneck cost in a system, there is little benefit from optimizing other smaller costs, such as the cost of concurrency control. However, in a main-memory database, with disk I/O no longer the bottleneck, systems benefit from reducing other costs, such as query processing costs, as we saw in Section 15.8; we now consider how to reduce the cost of concurrency control in main-memory databases.

As we saw in Section 18.10.2, concurrency-control techniques for operations on disk-based index structures acquire locks on individual nodes, to increase the potential for concurrent access to the index. However, such locking comes at the increased cost of acquiring the locks. In a main-memory database, where data are in memory, index operations take very little time for execution. Thus, it may be acceptable to perform locking at a coarse granularity: for example, the entire index could be locked using a single latch (i.e., short duration lock), the operation performed, and the latch released. The reduced overhead of locking has been found to make up for the slightly reduced concurrency, and to improve overall performance.

There is another way to improve performance with in-memory indices, using atomic instructions to carry out index updates without acquiring any latches at all.

Telegram: @uni_k

www.konkur.in


```
insert(value, head) {
        node = new node
        node->value = value
        node->next = head
        head = node
}
```

Figure 18.22 Insertion code that is unsafe with concurrent inserts.

Data structures implementations that support concurrent operations without requiring latches are called **latch-free data structure** implementations.

Consider a linked list, where each node has a value *value* and a *next* pointer, and the head of the linked list is stored in the variable *head*. The function *insert*() shown in Figure 18.22 would work correctly to insert a node at the head of the list, if there are no concurrent invocations of the code for the same list.[7]

However, if two processes execute the *insert*() function concurrently on the same list, it is possible that both of them would read the same value of variable *head*, and then both would update the variable after that. The final result would contain one of the two nodes being inserted, while the other node being inserted would be lost.

One way of preventing such a problem is to get an exclusive latch (short term lock) on the linked list, perform the *insert*() function, and then release the latch. The *insert*() function can be modified to acquire and release a latch on the list.

An alternative implementation, which is faster in practice, is to use an atomic *compare-and-swap*() instruction, abbreviated to CAS, which works as follows: The instruction CAS(*var*, *oldval*, *newval*) takes three arguments: a variable *var* and two values, *oldval* and *newval*. The instruction does the following atomically: check if the value of *var* is equal to *oldval*, and if so, set *var* to *newval*, and return success. If the value is not equal, it returns failure. The instruction is supported by most modern processor architectures, and it executes very quickly.

The function *insert_latchfree*(), shown in Figure 18.23 is a modification of *insert*() that works correctly even with concurrent inserts on the same list, without obtaining any latches. With this code, if two processes concurrently read the old value of *head*, and then both execute the CAS instruction, one of them will find the CAS instruction returning success, while the other one will find it returning failure since the value of *head* changes between the time it is read and when the CAS instruction is executed. The repeat loop then retries the insert using the new value of *head*, until it succeeds.

Function *delete_latchfree*(), shown in Figure 18.23, similarly implements deletion from the head of the list using the compare and swap instruction, without requiring latches. (In this case, the list is used as a stack, since deletion occurs at the head of

[7] We assume all parameters are passed by reference.

Telegram: @uni_k

www.konkur.in


```
insert_latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        oldhead = head
        node->next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}

delete_latchfree(head) {
    /* This function is not quite safe; see explanation in text. */
    repeat
        oldhead = head
        newhead = oldhead->next
        result = CAS(head, oldhead, newhead)
    until (result == success)
}
```

Figure 18.23 Latch-free insertion and deletion on a list.

the list.) However, it has a problem: it does not work correctly in some rare cases. The problem can occur when a process $P1$ is performing a delete, with node $n1$ at the head of the list, and concurrently a second process $P2$ deletes the first two elements, $n1$ and $n2$, and then reinserts $n1$ at the head of the list, with some other element, say $n3$ as the next element. If $P1$ read $n1$ before $P2$ deleted it, but performs the CAS after $P2$ has reinserted $n1$, the CAS operation of $P1$ will succeed, but set the head of the list to point to $n2$, which has been deleted, leaving the list in an inconsistent state. This problem is known as the *ABA problem*.

One solution is to keep a counter along with each pointer, which is incremented every time the pointer is updated. The CAS instruction is applied on the (pointer, counter) pair; most CAS implementations on 64 bit processors support such a double compare-and-swap on 128 bits. The ABA problem can then be avoided since although the reinsert of $n1$ would result in the head pointing to $n1$, the counter would be different, resulting in the CAS operation of $P1$ failing. See the online solutions to Practice Exercise 18.16 for more details of the ABA problem and the above solution. With such a modification, both inserts and deletes can be executed concurrently without acquiring latches. There are other solutions that do not require a double compare-and-swap, but are more complicated.

Deletion from the tail of the list (to implement a queue) as well as more complex data structures such as hash indices and search trees can also be implemented in a latch-

Telegram: @uni_k

www.konkur.in

free manner. It is best to use latch-free data structure implementations (more often referred to as **lock-free data structure** implementations) that are provided by standard libraries, such as the Boost library for C++, or the ConcurrentLinkedQueue class in Java; do not build your own, since you may introduce bugs due to "*race conditions*" between concurrent accesses, that can be very hard to detect or debug.

Since today's multiprocessor CPUs have a large number of cores, latch-free implementations have been found to significantly outperform implementations that obtain latches, in the context of in-memory indices and other in-memory data structures

18.10.4 Long-Duration Transactions

The transaction concept developed initially in the context of data-processing applications, in which most transactions are noninteractive and of short duration. Serious problems arise when this concept is applied to database systems that involve human interaction. Such transactions have these key properties:

- **Long duration**. Once a human interacts with an active transaction, that transaction becomes a **long-duration transaction** from the perspective of the computer, since human response time is slow relative to computer speed. Furthermore, in design applications, the human activity may involve hours, days, or an even longer period. Thus, transactions may be of long duration in human terms, as well as in machine terms.
- **Exposure of uncommitted data**. Data generated and displayed to a user by a long-duration transaction are uncommitted, since the transaction may abort. Thus, users—and, as a result, other transactions—may be forced to read uncommitted data. If several users are cooperating on a project, user transactions may need to exchange data prior to transaction commit.
- **Subtasks**. An interactive transaction may consist of a set of subtasks initiated by the user. The user may wish to abort a subtask without necessarily causing the entire transaction to abort.
- **Recoverability**. It is unacceptable to abort a long-duration interactive transaction because of a system crash. The active transaction must be recovered to a state that existed shortly before the crash so that relatively little human work is lost.
- **Performance**. Good performance in an interactive transaction system is defined as fast response time. This definition is in contrast to that in a noninteractive system, in which high throughput (number of transactions per second) is the goal. Systems with high throughput make efficient use of system resources. However, in the case of interactive transactions, the most costly resource is the user. If the efficiency and satisfaction of the user are to be optimized, response time should be fast (from a human perspective). In those cases where a task takes a long time, response time

Telegram: @uni_k

www.konkur.in

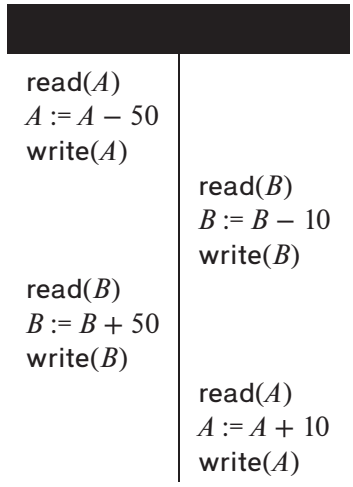


Figure 18.24 A non-conflict-serializable schedule.

should be predictable (i.e., the variance in response times should be low) so that users can manage their time well.

Snapshot isolation, described in Section 18.8, can provide a partial solution to these issues, as can the *optimistic concurrency control without read validation* protocol described in Section 18.9.3. The latter protocol was in fact designed specifically to deal with long-duration transactions that involve user interaction. Although it does not guarantee serializability, optimistic concurrency control without read validation is quite widely used.

However, when transactions are of long duration, conflicting updates are more likely, resulting in additional waits or aborts. These considerations are the basis for the alternative concepts of correctness of concurrent executions and transaction recovery that we consider in the remainder of this section.

18.10.5 Concurrency Control with Operations

Consider a bank database consisting of two accounts A and B, with the consistency requirement that the sum $A + B$ be preserved. Consider the schedule of Figure 18.24. Although the schedule is not conflict serializable, it nevertheless preserves the sum of $A + B$. It also illustrates two important points about the concept of correctness without serializability.

1. Correctness depends on the specific consistency constraints for the database.
2. Correctness depends on the properties of operations performed by each transaction.

Telegram: @uni_k

www.konkur.in

While two-phase locking ensures serializability, it can result in poor concurrency in case a large number of transactions conflict on a particular data item. Timestamp and validation protocols also have similar problems in this case.

Concurrency can be increased by treating some operations besides **read** and **write** as fundamental low-level operations and to extend concurrency control to deal with them.

Consider the case of materialized view maintenance, which we saw in Section 16.5.1. Suppose there is a relation *sales*(*date, custID, itemID, amount*), and a materialized view *daily_sales_total*(*date, total_amount*), that records total sales on each day. Every sales transaction must update the materialized view as part of the transaction if immediate view maintenance is used. With a high volume of sales, and every transaction updating the same record in the *daily_sales_total* relation, the degree of concurrency will be quite low if two-phase locking is used on the materialized view.

A better way to perform concurrency control for the materialized view is as follows: Observe that each transaction increments a record in the *daily_sales_total* relation by some value but does not need to see the value. It would make sense to have an operation increment(v, n), that adds a value n to a variable v without making the value of v visible to the transaction; we shall see shortly how this is implemented. In our sales example, a transaction that inserts a *sales* tuple with amount n invokes the increment operation with the first argument being the *total_amount* value of the appropriate tuple in the materialized view *daily_sales_total*, and the second argument being the value n.

The increment operation does not lock the variable in a two-phase manner; however, individual operations should be executed serially on the variable. Thus, if two increment operations are initiated concurrently on the same variable, one must finish before the other is allowed to start. This can be ensured by acquiring an exclusive latch (lock) on the variable v before starting the operation and releasing the latch after the operation has finished its updates. Increment operations can also be implemented using compare-and-swap operations, without getting latches.

Two transactions that invoke the increment operation should be allowed to execute concurrently to avoid concurrency control bottlenecks. In fact, increment operations executed by two transactions do not conflict with each other, since the final result is the same regardless of the order in which the operations were executed. If one of the transactions rolls back, the increment(v, n) operation must be rolled back by executing an operation increment($v, -n$), which adds a negative of the original value; this operation is referred to as a **compensating operation**.

However, if a transaction T wishes to read the materialized view, it clearly conflicts with any concurrent transaction that has performed an increment operation; the value that T reads depends on whether the other transaction is serialized before or after T.

We can define a locking protocol to handle the preceding situation by defining an **increment lock**. The increment lock is compatible with itself but is not compatible with shared and exclusive locks. Figure 18.25 shows a lock-compatibility matrix for three lock modes: share mode, exclusive mode, and increment mode.

Telegram: @uni_k

www.konkur.in

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Figure 18.25 Lock-compatibility matrix with increment lock mode.

As another example of special-purpose concurrency control for operations, consider an insert operation on a B^+-tree index which releases locks early, as we saw in Section 18.10.2. In this case, there is no special lock mode, but holding locks on leaf nodes in a two-phase manner (or using next-key locking) as we saw in Section 18.10.2 ensures serializability. The insert operation may have modified several nodes of the B^+-tree index. Other transactions may have read and updated these nodes further while processing other operations. To roll back the insertion, we would have to delete the record inserted by T_i; deletion is the compensating action for insertion. The result is a correct, consistent B^+-tree, but not necessarily one with exactly the same structure as the one we had before T_i started.

While operation locking can be done in a way that ensures serializability, in some cases it may even be used in a way that does not guarantee serializability, but where violations may be acceptable. Consider the case of concert tickets, where every transaction needs to access and update the total ticket sales. We can have an operation increment_conditional(v, n) which increments v by n, provided the resultant value would be ≥ 0; the operation returns a status of success in case the resultant value is ≥ 0 and returns failure otherwise. Consider a transaction T_i executed to purchase tickets. To book three tickets, where variable *avail_tickets* indicates the number of available tickets, the transaction can execute increment_conditional(*avail_tickets*, -3). A return value of success indicates that there were enough tickets available, and decrements the available tickets, while failure indicates insufficient availability of tickets.

If the variable *avail_tickets* is locked in a two-phase manner, concurrency would be very poor, with customers being forced to wait for bookings while an earlier transaction commits, even when there are many tickets available. Concurrency can be greatly increased by executing the increment_conditional operation, without holding any locks on *avail_tickets* in a two-phase manner; instead, an exclusive lock is obtained on the variable, the operation is performed, and the lock is then released.

The transaction T_i also needs to carry out other steps, such as collecting the payment; if one of the subsequent steps, such as payment, fails, the increment operation must be rolled back by executing a compensating operation; if the original operation added $-n$ to *avail_tickets*, the compensating operation adds $+n$ to *avail_tickets*.

It may appear that two increment_conditional operations are compatible with each other, similar to the increment operation that we saw earlier. But that is not

Telegram: @uni_k

www.konkur.in

the case. Consider two concurrent transactions to purchase a single ticket, and assume that there is only one ticket left. The order in which the operations are executed has an obvious impact on which one succeeds and which one fails. Nevertheless, many real-world applications allow operations that hold short-term locks while they execute and release them at the end of the operation to increase concurrency, even at the cost of loss of serializability in some situations.

18.10.6 Real-Time Transaction Systems

In certain applications, the constraints include **deadlines** by which a task must be completed. Examples of such applications include plant management, traffic control, and scheduling. When deadlines are included, correctness of an execution is no longer solely an issue of database consistency. Rather, we are concerned with how many deadlines are missed, and by how much time they are missed. Deadlines are characterized as follows:

- **Hard deadline**. Serious problems, such as system crash, may occur if a task is not completed by its deadline.
- **Firm deadline**. The task has zero value if it is completed after the deadline.
- **Soft deadlines**. The task has diminishing value if it is completed after the deadline, with the value approaching zero as the degree of lateness increases.

Systems with deadlines are called **real-time systems**.

Transaction management in real-time systems must take deadlines into account. If the concurrency-control protocol determines that a transaction T_i must wait, it may cause T_i to miss the deadline. In such cases, it may be preferable to pre-empt the transaction holding the lock, and to allow T_i to proceed. Pre-emption must be used with care, however, because the time lost by the pre-empted transaction (due to rollback and restart) may cause the pre-empted transaction to miss its deadline. Unfortunately, it is difficult to determine whether rollback or waiting is preferable in a given situation.

Due to the unpredictable nature of delays when reading data from disk, main-memory databases are often used if real-time constraints have to be met. However, even if data are resident in main memory, variances in execution time arise from lock waits, transaction aborts, and so on. Researchers have devoted considerable effort to concurrency control for real-time databases. They have extended locking protocols to provide higher priority for transactions with early deadlines. They have found that optimistic concurrency protocols perform well in real-time databases; that is, these protocols result in fewer missed deadlines than even the extended locking protocols. The online bibliographical notes provide references to research in the area of real-time databases.

18.11 Summary

- When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary

Telegram: @uni_k

www.konkur.in

teraction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.

- To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes.
- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.
- The strict two-phase locking protocol permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness of the resulting schedules. The rigorous two-phase locking protocol releases all locks only at the end of the transaction.
- Various locking protocols do not guard against deadlocks. One way to prevent deadlock is to use an ordering of data items and to request locks in a sequence consistent with the ordering.
- Another way to prevent deadlock is to use preemption and transaction rollbacks. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps to decide whether a transaction should wait or roll back. The wound – wait scheme is a preemptive scheme.
- If deadlocks are not prevented, the system must deal with them by using a deadlock detection and recovery scheme. To do so, the system constructs a wait-for graph. A system is in a deadlock state if and only if the wait-for graph contains a cycle. When the deadlock detection algorithm determines that a deadlock exists, the system rolls back one or more transactions to break the deadlock.
- There are circumstances where it would be advantageous to group several data items and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and we define a hierarchy of data items where the small items are nested within larger ones. Such a hierarchy can be represented graphically as a tree. In such multigranularity locking protocols, locks are acquired in root-to-leaf order; they are released in leaf-to-root order. Intention lock modes are used at higher levels to get better concurrency, without affecting serializability.

Telegram: @uni_k

www.konkur.in

- A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction T_i is smaller than the timestamp of transaction T_j, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j. It does so by rolling back a transaction whenever such an order is violated.
- A validation scheme is an appropriate concurrency-control method in cases where a majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions is low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test to complete. If it does not pass the validation test, the system rolls it back to its initial state.
- A multiversion concurrency-control scheme is based on the creation of a new version of a data item for each transaction that writes that item. When a read operation is issued, the system selects one of the versions to be read. The concurrency-control scheme ensures that the version to be read is selected in a manner that ensures serializability by using timestamps. A read operation always succeeds.
    - In multiversion timestamp ordering, a write operation may result in the rollback of the transaction.
    - In multiversion two-phase locking, write operations may result in a lock wait or, possibly, in deadlock.
- Snapshot isolation is a multiversion concurrency-control protocol based on validation, which, unlike multiversion two-phase locking, does not require transactions to be declared as read-only or update. Snapshot isolation does not guarantee serializability but is nevertheless supported by many database systems. Serializable snapshot isolation is an extension of snapshot isolation which guarantees serializability.
- A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted. A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.
- Insertions can lead to the phantom phenomenon, in which an insertion logically conflicts with a query even though the two transactions may access no tuple in common. Such conflict cannot be detected if locking is done only on tuples accessed by the transactions. Locking is required on the data used to find the tuples in the relation. The index-locking technique solves this problem by requiring locks on certain index nodes. These locks ensure that all conflicting transactions conflict on a real data item, rather than on a phantom.

Telegram: @uni_k

www.konkur.in

- Weak levels of consistency are used in some applications where consistency of query results is not critical, and using serializability would result in queries adversely affecting transaction processing. Degree-two consistency is one such weaker level of consistency; cursor stability is a special case of degree-two consistency and is widely used.
- Concurrency control is a challenging task for transactions that span user interactions. Applications often implement a scheme based on validation of writes using version numbers stored in tuples; this scheme provides a weak level of serializability and can be implemented at the application level without modifications to the database.
- Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in B^+-trees to allow greater concurrency. These techniques allow nonserializable access to the B^+-tree, but they ensure that the B^+-tree structure is correct, and they ensure that accesses to the database itself are serializable. Latch-free data structures are used to implement high-performance indices and other data structures in main-memory databases.

Review Terms

- Concurrency control
- Lock types
  - Shared-mode (S) lock
  - Exclusive-mode (X) lock
- Lock
  - Compatibility
  - Request
  - Wait
  - Grant
- Deadlock
- Starvation
- Locking protocol
- Legal schedule
- Two-phase locking protocol
  - Growing phase
  - Shrinking phase
  - Lock point
  - Strict two-phase locking
  - Rigorous two-phase locking
- Lock conversion
  - Upgrade
  - Downgrade
- Graph-based protocols
  - Tree protocol
  - Commit dependency
- Deadlock handling
  - Prevention
  - Detection
  - Recovery
- Deadlock prevention
  - Ordered locking
  - Preemption of locks

Telegram: @uni_k

www.konkur.in

- Wound–wait scheme
- Timeout-based schemes

- Deadlock detection
  - Wait-for graph
- Deadlock recovery
  - Total rollback
  - Partial rollback
- Multiple granularity
  - Explicit locks
  - Implicit locks
  - Intention locks
- Intention lock modes
  - Intention-shared (IS)
  - Intention-exclusive (IX)
  - Shared and intention-exclusive (SIX)
- Multiple-granularity locking protocol
- Timestamp
  - System clock
  - Logical counter
  - W-timestamp(Q)
  - R-timestamp(Q)
- Timestamp-ordering protocol
  - Thomas' write rule
- Validation-based protocols
  - Read phase
  - Validation phase
  - Write phase
  - Validation test
- Multiversion timestamp ordering
- Multiversion two-phase locking
  - Read-only transactions
  - Update transactions
- Snapshot isolation
  - Lost update
  - First committer wins
  - First updater wins
  - Write skew
  - Select for update
- Insert and delete operations
- Phantom phenomenon
- Index-locking protocol
- Predicate locking
- Weak levels of consistency
  - Degree-two consistency
  - Cursor stability
- Optimistic concurrency control without read validation
- Conversations
- Concurrency in indices
  - Crabbing protocol
  - B-link trees
  - B-link-tree locking protocol
  - Next-key locking
- Latch-free data structures
- Compare-and-swap (CAS) instruction

Telegram: @uni_k

www.konkur.in

Practice Exercises

18.1 Show that the two-phase locking protocol ensures conflict serializability and that transactions can be serialized according to their lock points.

18.2 Consider the following two transactions:

T_{34}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Add lock and unlock instructions to transactions T_{31} and T_{32} so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

18.3 What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

18.4 Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

18.5 Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

18.6 Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are accessed. Most modern operating systems allow the user to set access protections (no access, read, write) on pages, and memory access that violate the access protections result in a protection violation (see the Unix `mprotect` command, for example). Describe how the access-protection mechanism can be used for page-level locking in a persistent programming language.

18.7 Consider a database system that includes an atomic **increment** operation, in addition to the read and write operations. Let V be the value of data item X. The operation

increment(X) by C

Telegram: @uni_k

www.konkur.in

sets the value of X to $V + C$ in an atomic step. The value of X is not available to the transaction unless the latter executes a read(X).
Assume that increment operations lock the item in increment mode using the compatibility matrix in Figure 18.25.

a. Show that, if all transactions lock the data that they access in the corresponding mode, then two-phase locking ensures serializability.

b. Show that the inclusion of **increment** mode locks allows for increased concurrency.

18.8 In timestamp ordering, **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write(Q) successfully. Would this change in wording make any difference? Explain your answer.

18.9 Use of multiple-granularity locking may require more or fewer locks than an equivalent system with a single lock granularity. Provide examples of both situations, and compare the relative amount of concurrency allowed.

18.10 For each of the following protocols, describe aspects of practical applications that would lead you to suggest using the protocol, and aspects that would suggest not using the protocol:

- Two-phase locking
- Two-phase locking with multiple-granularity locking.
- The tree protocol
- Timestamp ordering
- Validation
- Multiversion timestamp ordering
- Multiversion two-phase locking

18.11 Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation-based techniques, but unlike the validation techniques do not perform either validation or writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O.)

18.12 Consider the timestamp-ordering protocol, and two transactions, one that writes two data items p and q, and another that reads the same two data items.

Telegram: @uni_k

www.konkur.in

Give a schedule whereby the timestamp test for a write operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a **livelock**.)

18.13 Devise a timestamp-based protocol that avoids the phantom phenomenon.

18.14 Suppose that we use the tree protocol of Section 18.1.5 to manage concurrent access to a B^+-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

18.15 The snapshot isolation protocol uses a validation step which, before performing a write of a data item by transaction T, checks if a transaction concurrent with T has already written the data item.

a. A straightforward implementation uses a start timestamp and a commit timestamp for each transaction, in addition to an *update set*, that, is the set of data items updated by the transaction. Explain how to perform validation for the first-committer-wins scheme by using the transaction timestamps along with the update sets. You may assume that validation and other commit processing steps are executed serially, that is, for one transaction at a time,

b. Explain how the validation step can be implemented as part of commit processing for the first-committer-wins scheme, using a modification of the above scheme, where instead of using update sets, each data item has a write timestamp associated with it. Again, you may assume that validation and other commit processing steps are executed serially.

c. The first-updater-wins scheme can be implemented using timestamps as described above, except that validation is done immediately after acquiring an exclusive lock, instead of being done at commit time.

   i. Explain how to assign write timestamps to data items to implement the first-updater-wins scheme.

   ii. Show that as a result of locking, if the validation is repeated at commit time the result would not change.

   iii. Explain why there is no need to perform validation and other commit processing steps serially in this case.

18.16 Consider functions *insert_latchfree*() and *delete_latchfree*(), shown in Figure 18.23.

Telegram: @uni_k

www.konkur.in

a. Explain how the ABA problem can occur if a deleted node is reinserted.

b. Suppose that adjacent to *head* we store a counter *cnt*. Also suppose that DCAS((*head,cnt*), (*oldhead, oldcnt*), (*newhead, newcnt*)) atomically performs a compare-and-swap on the 128 bit value (*head,cnt*). Modify the *insert_latchfree*( ) and *delete_latchfree*( ) to use the DCAS operation to avoid the ABA problem.

c. Since most processors use only 48 bits of a 64 bit address to actually address memory, explain how the other 16 bits can be used to implement a counter, in case the DCAS operation is not supported.

Exercises

18.17 What benefit does strict two-phase locking provide? What disadvantages result?

18.18 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

18.19 Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction T_i must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by T_i after it has been unlocked by T_i.

Show that the forest protocol does *not* ensure serializability.

18.20 Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?

18.21 If deadlock is avoided by deadlock-avoidance schemes, is starvation still possible? Explain your answer.

18.22 In multiple-granularity locking, what is the difference between implicit and explicit locking?

18.23 Although SIX mode is useful in multiple-granularity locking, an exclusive and intention-shared (XIS) mode is of no use. Why is it useless?

18.24 The multiple-granularity protocol rules specify that a transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either

Telegram: @uni_k

www.konkur.in

IX or IS mode. Given that SIX and S locks are stronger than IX or IS locks, why does the protocol not allow locking a node in S or IS mode if the parent is locked in either SIX or S mode?

18.25 Suppose the lock hierarchy for a database consists of database, relations, and tuples.

a. If a transaction needs to read a lot of tuples from a relation r, what locks should it acquire?

b. Now suppose the transaction wants to update a few of the tuples in r after reading a lot of tuples. What locks should it acquire?

c. If at run-time the transaction finds that it needs to actually update a very large number of tuples (after acquiring locks assuming only a few tuples would be updated). What problems would this cause to the lock table, and what could the database do to avoid the problem?

18.26 When a transaction is rolled-back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?

18.27 Show that there are schedules that are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

18.28 Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a `read` request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for `write` requests?

18.29 As discussed in Exercise 18.15, snapshot isolation can be implemented using a form of timestamp validation. However, unlike the multiversion timestamp-ordering scheme, which guarantees serializability, snapshot isolation does not guarantee serializability. Explain the key difference between the protocols that results in this difference.

18.30 Outline the key similarities and differences between the timestamp-based implementation of the first-committer-wins version of snapshot isolation, described in Exercise 18.15, and the optimistic-concurrency control-without-read-validation scheme, described in Section 18.9.3.

18.31 Consider a relation $r(A, B, C)$ and a transaction T that does the following: find the maximum A value in r, and insert a new tuple in r whose A value is 1+ the maximum A value. Assume that an index is used to find the maximum A value.

a. Suppose that the transaction locks each tuple it reads in S mode, and the tuple it creates in X mode, and performs no other locking. Now suppose two instances of T are run concurrently. Explain how the resultant execution could be non-serializable.

Telegram: @uni_k

www.konkur.in

b. Now suppose that $r.A$ is declared as a primary key. Can the above nonserializable execution occur in this case? Explain why or why not.

18.32 Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?

18.33 Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?

18.34 Give example schedules to show that with key-value locking, if lookup, insert, or delete does not lock the next-key value, the phantom phenomenon could go undetected.

18.35 Many transactions update a common item (e.g., the cash balance at a branch) and private items (e.g., individual account balances). Explain how you can increase concurrency (and throughput) by ordering the operations of the transaction.

18.36 Consider the following locking protocol: All items are numbered, and once an item is unlocked, only higher-numbered items may be locked. Locks may be released at any time. Only X-locks are used. Show by an example that this protocol does not guarantee serializability.

Further Reading

[Gray and Reuter (1993)] provides detailed textbook coverage of transaction-processing concepts, including concurrency-control concepts and implementation details. [Bernstein and Newcomer (2009)] provides textbook coverage of various aspects of transaction processing including concurrency control.

The two-phase locking protocol was introduced by [Eswaran et al. (1976)]. The locking protocol for multiple-granularity data items is from [Gray et al. (1975)]. The timestamp-based concurrency-control scheme is from [Reed (1983)]. The validation concurrency-control scheme is from [Kung and Robinson (1981)]. Multiversion timestamp order was introduced in [Reed (1983)]. A multiversion tree-locking algorithm appears in [Silberschatz (1982)].

Degree-two consistency was introduced in [Gray et al. (1975)]. The levels of consistency—or isolation—offered in SQL are explained and critiqued in [Berenson et al. (1995)]; the snapshot isolation technique was also introduced in the same paper. Serializable snapshot-isolation was introduced by [Cahill et al. (2009)]; [Ports and Grittner (2012)] describes the implementation of serializable snapshot isolation in PostgreSQL.

Concurrency in B^+-trees was studied by [Bayer and Schkolnick (1977)] and [Johnson and Shasha (1993)]. The crabbing and B-link tree techniques were introduced by [Kung and Lehman (1980)] and [Lehman and Yao (1981)]. The technique of key-value locking used in ARIES provides for very high concurrency on B^+-tree access and is de-

Telegram: @uni_k

www.konkur.in

scribed in [Mohan (1990)] and [Mohan and Narang (1992)]. [Faerber et al. (2017)] provide a survey of main-memory databases, including coverage of concurrency control in main-memory databases. The ABA problem with latch-free data structures as well as solutions for the problem are discussed in [Dechev et al. (2010)].

Bibliography

[Bayer and Schkolnick (1977)] R. Bayer and M. Schkolnick, "Concurrency of Operating on B-trees", *Acta Informatica*, Volume 9, Number 1 (1977), pages 1–21.

[Berenson et al. (1995)] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 1–10.

[Bernstein and Newcomer (2009)] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd edition, Morgan Kaufmann (2009).

[Cahill et al. (2009)] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases", *ACM Transactions on Database Systems*, Volume 34, Number 4 (2009), pages 20:1–20:42.

[Dechev et al. (2010)] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs", In *IEEE Int'l Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing, (ISORC)* (2010), pages 185–192.

[Eswaran et al. (1976)] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Volume 19, Number 11 (1976), pages 624–633.

[Faerber et al. (2017)] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo, "Main Memory Database Systems", *Foundations and Trends in Databases*, Volume 8, Number 1-2 (2017), pages 1–130.

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Gray et al. (1975)] J. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", In *Proc. of the International Conf. on Very Large Databases* (1975), pages 428–451.

[Johnson and Shasha (1993)] T. Johnson and D. Shasha, "The Performance of Concurrent B-Tree Algorithms", *ACM Transactions on Database Systems*, Volume 18, Number 1 (1993), pages 51–101.

[Kung and Lehman (1980)] H. T. Kung and P. L. Lehman, "Concurrent Manipulation of Binary Search Trees", *ACM Transactions on Database Systems*, Volume 5, Number 3 (1980), pages 339–353.

Telegram: @uni_k

www.konkur.in

[Kung and Robinson (1981)] H. T. Kung and J. T. Robinson, "Optimistic Concurrency Control", *ACM Transactions on Database Systems*, Volume 6, Number 2 (1981), pages 312–326.

[Lehman and Yao (1981)] P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, Volume 6, Number 4 (1981), pages 650–670.

[Mohan (1990)] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operations on B-Tree indexes", In *Proc. of the International Conf. on Very Large Databases* (1990), pages 392–405.

[Mohan and Narang (1992)] C. Mohan and I. Narang, "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment", In *Proc. of the International Conf. on Extending Database Technology* (1992), pages 453–468.

[Ports and Grittner (2012)] D. R. K. Ports and K. Grittner, "Serializable Snapshot Isolation in PostgreSQL", *Proceedings of the VLDB Endowment*, Volume 5, Number 12 (2012), pages 1850–1861.

[Reed (1983)] D. Reed, "Implementing Atomic Actions on Decentralized Data", *Transactions on Computer Systems*, Volume 1, Number 1 (1983), pages 3–23.

[Silberschatz (1982)] A. Silberschatz, "A Multi-Version Concurrency Control Scheme With No Rollbacks", In *Proc. of the ACM Symposium on Principles of Distributed Computing* (1982), pages 216–223.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 19



Recovery System

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Chapter 17, are preserved. An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.

The recovery scheme must also support **high availability**, that is, the database should be usable for a very high percentage of time. To support high availability in the face of machine failure (as also planned machine shutdowns for hardware/software upgrades and maintenance), the recovery scheme must support the ability to keep a backup copy of the database synchronized with the current contents of the primary copy of the database. If the machine with the primary copy fails, transaction processing can continue on the backup copy.

19.1 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. In this chapter, we shall consider only the following types of failure:

- **Transaction failure**. There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (e.g., deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.

Telegram: @uni_k

www.konkur.in

- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage and brings transaction processing to a halt. The content of non-volatile storage remains intact and is not corrupted.

  The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the non-volatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

19.2 Storage

As we saw in Chapter 13, the various data items in the database may be stored and accessed in a number of different storage media. In Section 17.3, we saw that storage media can be distinguished by their relative speed, capacity, and resilience against failure. We identified three categories of storage:

1. **Volatile storage**
2. **Non-Volatile storage**
3. **Stable storage**

Stable storage or, more accurately, an approximation thereof, plays a critical role in recovery algorithms.

Telegram: @uni_k

www.konkur.in

19.2.1 Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several non-volatile storage media (usually disk) with independent failure modes and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall (from Chapter 12) that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off-site to guard against such disasters. However, since tapes cannot be carried off-site continually, updates since the most recent time that tapes were carried off-site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood. We study such *remote backup* systems in Section 19.7.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in:

- **Successful completion**. The transferred information arrived safely at its destination.
- **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.

Telegram: @uni_k

www.konkur.in

If the system fails while blocks are being written, it is possible that the two copies of a block could be inconsistent with each other. During recovery, for each block, the system would need to examine two copies of the blocks. If both are the same and no detectable error exists, then no further actions are necessary. (Recall that errors in a disk block, such as a partial write to the block, are detected by storing a checksum with each block.) If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system can either replace the content of the first block with the value of the second, or replace the content of the second block with the value of the first. Either way, the recovery procedure ensures that a write to stable storage either succeeds completely (i.e., updates all copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of non-volatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

The protocols for writing out a block to a remote site are similar to the protocols for writing blocks to a mirrored disk system, which we examined in Chapter 12, and particularly in Practice Exercise 12.6.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

19.2.2 Data Access

As we saw in Chapter 12, the database system resides permanently on non-volatile storage (usually disks), and only parts of the database are in memory at any time. (In main-memory databases, the entire database resides in memory, but a copy still resides on non-volatile storage so data can survive the loss of main-memory contents.) The database is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer to and from disk and may contain several data items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as a bank or a university.

Transactions input information from the disk into main memory and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following two operations:

1. input(B) transfers the physical block B to main memory.

Telegram: @uni_k

www.konkur.in

2. output(B) transfers the buffer block B to the disk and replaces the appropriate physical block there.

Figure 19.1 illustrates this scheme.

Conceptually, each transaction T_i has a private work area in which copies of data items accessed and updated by T_i are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each data item X kept in the work area of transaction T_i is denoted by x_i. Transaction T_i interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

1. read(X) assigns the value of data item X to the local variable x_i. It executes this operation as follows:
   a. If block B_X on which X resides is not in main memory, it issues input(B_X).
   b. It assigns to x_i the value of X from the buffer block.
2. write(X) assigns the value of local variable x_i to data item X in the buffer block. It executes this operation as follows:
   a. If block B_X on which X resides is not in main memory, it issues input(B_X).
   b. It assigns the value of x_i to X in buffer B_X.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes

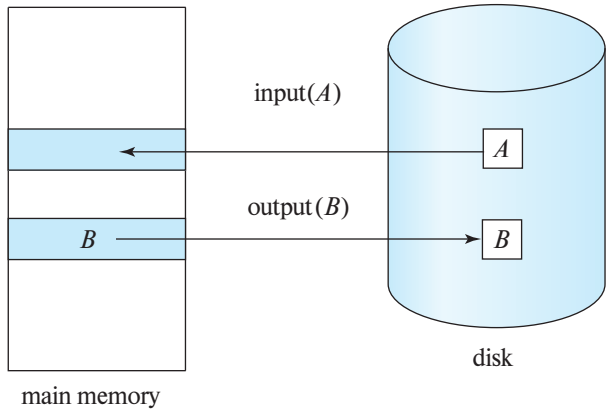


Figure 19.1 Block storage operations.

Telegram: @uni_k

www.konkur.in

to reflect the change to B on the disk. We shall say that the database system performs a **force-output** of buffer B if it issues an output(B).

When a transaction needs to access a data item X for the first time, it must execute read(X). The transaction then performs all updates to X on x_i. At any point during its execution a transaction may execute write(X) to reflect the change to X in the database itself; write(X) must certainly be done after the final write to x_i.

The output(B_X) operation for the buffer block B_X on which X resides does not need to take effect immediately after write(X) is executed, since the block B_X may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write(X) operation was executed but before output(B_X) was executed, the new value of X is never written to disk and, thus, is lost. As we shall see shortly, the database system executes extra actions to ensure that updates performed by committed transactions are not lost even if there is a system crash.

19.3 Recovery and Atomicity

Consider again our simplified banking system and a transaction T_i that transfers \$50 from account A to account B, with initial values of A and B being \$1000 and \$2000, respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction.

When the system restarts, the value of A would be \$950, while that of B would be \$2000, which is clearly inconsistent with the atomicity requirement for transaction T_i. Unfortunately, there is no way to find out by examining the database state what blocks had been output and what had not before the crash. It is possible that the transaction completed, updating the database on stable storage from an initial state with the values of A and B being \$1000 and \$1950; it is also possible that the transaction did not affect the stable storage at all, and the values of A and B were \$950 and \$2000 initially; or that the updated B was output but not the updated A; or that the updated A was output but the updated B was not.

Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output to stable storage information describing the modifications, without modifying the database itself. As we shall see, this information can help us ensure that all modifications performed by committed transactions are reflected in the database (perhaps during the course of recovery actions after a crash). We also need to store information about the old value of any item updated by a modification in case the transaction performing the modification

Telegram: @uni_k

www.konkur.in

fails (aborts). This information can help us undo the modifications made by the failed transaction.

The most commonly used technique for recovery is based on log records, and we study **log-based recovery** in detail in this chapter. An alternative, called shadow copying, is used by text editors but is not used in database systems; this approach is summarized in Note 19.1 on page 914.

19.3.1 Log Records

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database.

There are several types of log records. An **update log record** describes a single database write. It has these fields:

- **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides and an offset within the block.
- **Old value**, which is the value of the data item prior to the write.
- **New value**, which is the value that the data item will have after the write.

We represent an update log record as $<T_i, X_j, V_1, V_2>$, indicating that transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write and has value V_2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

- $<T_i$ start$>$. Transaction T_i has started.
- $<T_i$ commit$>$. Transaction T_i has committed.
- $<T_i$ abort$>$. Transaction T_i has aborted.

We shall introduce several other types of log records later.

Whenever a transaction performs a write, it is essential that the log record for that write be created and added to the log, before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end

Telegram: @uni_k

www.konkur.in



of the log on stable storage as soon as it is created. In Section 19.5, we shall see when it is safe to relax this requirement so as to reduce the overhead imposed by logging. Observe that the log contains a complete record of all database activity. As a result, the volume of data stored in the log may become unreasonably large. In Section 19.3.6, we shall show when it is safe to erase log information.

Telegram: @uni_k

www.konkur.in

19.3.2 Database Modification

As we noted earlier, a transaction creates a log record prior to modifying the database. The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk. In order for us to understand the role of these log records in recovery, we need to consider the steps a transaction takes in modifying a data item:

1. The transaction performs some computations in its own private part of main memory.
2. The transaction modifies the data block in the disk buffer in main memory holding the data item.
3. The database system executes the output operation that writes the data block to disk.

We say a transaction *modifies the database* if it performs an update on a disk buffer, or on the disk itself; updates to the private part of main memory do not count as database modifications. If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique. If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique. Deferred modification has the overhead that transactions need to make local copies of all updated data items; further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

The recovery algorithms we describe in this chapter support immediate modification. As described, they work correctly even with deferred modification, but they can be optimized to reduce overhead when used with deferred modification; we leave details as an exercise.

A recovery algorithm must take into account a variety of factors, including:

- The possibility that a transaction may have committed although some of its database modifications exist only in the disk buffer in main memory and not in the database on disk.
- The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and the new value that is to be written for the data item. This allows the system to perform *undo* and *redo* operations as appropriate.

Telegram: @uni_k

www.konkur.in

- The **undo** operation using a log record sets the data item specified in the log record to the old value contained in the log record.
- The **redo** operation using a log record sets the data item specified in the log record to the new value contained in the log record.

19.3.3 Concurrency Control and Recovery

If the concurrency control scheme allows a data item X that has been modified by a transaction T_1 to be further modified by another transaction T_2 before T_1 commits, then undoing the effects of T_1 by restoring the old value of X (before T_1 updated X) would also undo the effects of T_2. To avoid such situations, recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts.

This requirement can be ensured by acquiring an exclusive lock on any updated data item and holding the lock until the transaction commits; in other words, by using strict two-phase locking. Snapshot isolation and validation-based concurrency-control techniques also acquire exclusive locks on data items at the time of validation, before modifying the data items, and hold the locks until the transaction is committed; as a result the above requirement is satisfied even by these concurrency control protocols.

We discuss in Section 19.8 how the above requirement can be relaxed in certain cases.

When either snapshot isolation or validation is used for concurrency control, database updates of a transaction are (conceptually) deferred until the transaction is partially committed; the deferred-modification technique is a natural fit with these concurrency control schemes. However, it is worth noting that some implementations of snapshot isolation use immediate modification but provide a logical snapshot on demand: when a transaction needs to read an item that a concurrent transaction has updated, a copy of the (already updated) item is made, and updates made by concurrent transactions are rolled back on the copy of the item. Similarly, immediate modification of the database is a natural fit with two-phase locking, but deferred modification can also be used with two-phase locking.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Figure 19.2 Portion of the system log corresponding to T and T.

Telegram: @uni_k

www.konkur.in

19.3.4 Transaction Commit

We say that a transaction has **committed** when its commit log record, which is the last log record of the transaction, has been output to stable storage; at that point all earlier log records have already been output to stable storage. Thus, there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone. If a system crash occurs before a log record $<T_i$ commit$>$ is output to stable storage, transaction T_i will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed.[1]

With most log-based recovery techniques, including the ones we describe in this chapter, blocks containing the data items modified by a transaction do not have to be output to stable storage when the transaction commits but can be output some time later. We discuss this issue further in Section 19.5.2.

19.3.5 Using the Log to Redo and Undo Transactions

We now provide an overview of how the log can be used to recover from a system crash and to roll back transactions during normal operation. However, we postpone details of the procedures for failure recovery and rollback to Section 19.4.

Consider our simplified banking system. Let T_0 be a transaction that transfers \$50 from account A to account B:

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let T_1 be a transaction that withdraws \$100 from account C:

T_1: read(C);
$C := C - 100$;
write(C).

The portion of the log containing the relevant information concerning these two transactions appears in Figure 19.2.

Figure 19.3 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T_0 and T_1.[2]

[1] The output of a block can be made atomic by techniques for dealing with data-transfer failure, as described in Section 19.2.1.

[2] Notice that this order could not be obtained using the deferred-modification technique, because the database is modified by T_0 before it commits, and likewise for T_1.

Telegram: @uni_k

www.konkur.in

Log	Database
$<T_0$ start$>$	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0$ commit$>$	
$<T_1$ start$>$	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1$ commit$>$	

Figure 19.3 State of system log and database corresponding to T_0 and T_1.

Using the log, the system can handle any failure that does not result in the loss of information in non-volatile storage. The recovery scheme uses two recovery procedures. Both these procedures make use of the log to find the set of data items updated by each transaction T_i and their respective old and new values.

- redo(T_i). The procedure sets the value of all data items updated by transaction T_i to the new values. The order in which updates are carried out by redo is important; when recovering from a system crash, if updates to a particular data item are applied in an order different from the order in which they were applied originally, the final state of that data item will have a wrong value. Most recovery algorithms, including the one we describe in Section 19.4, do not perform redo of each transaction separately; instead they perform a single scan of the log, during which redo actions are performed for each log record as it is encountered. This approach ensures the order of updates is preserved, and it is more efficient since the log needs to be read only once overall, instead of once per transaction.
- undo(T_i). The procedure restores the value of all data items updated by transaction T_i to the old values. In the recovery scheme that we describe in Section 19.4:
  - The undo operation not only restores the data items to their old value, but also writes log records to record the updates performed as part of the undo process. These log records are special **redo-only** log records, since they do not need to contain the old value of the updated data item; note that when such log records are used during undo, the "old value" is actually the value written by the transaction that is being rolled back, and the "new value" is the original value that is being restored by the undo operation.

    As with the redo procedure, the order in which undo operations are performed is important; again we postpone details to Section 19.4.

Telegram: @uni_k

www.konkur.in

- When the undo operation for transaction T_i completes, it writes a <T_i abort> log record, indicating that the undo has completed.

  As we shall see in Section 19.4, the undo(T_i) procedure is executed only once for a transaction, if the transaction is rolled back during normal processing or if on recovering from a system crash, neither a commit nor an abort record is found for transaction T_i. As a result, every transaction will eventually have either a commit or an abort record in the log.

After a system crash has occurred, the system consults the log to determine which transactions need to be redone and which need to be undone so as to ensure atomicity.

- Transaction T_i needs to be undone if the log contains the record <T_i start> but does not contain either the record <T_i commit> or the record <T_i abort>.
- Transaction T_i needs to be redone if the log contains the record <T_i start> and either the record <T_i commit> or the record <T_i abort>. It may seem strange to redo T_i if the record <T_i abort> is in the log. To see why this works, note that if <T_i abort> is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo T_i's modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time.

As an illustration, return to our banking example, with transaction T_0 and T_1 executed one after the other in the order T_0 followed by T_1. Suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases appears in Figure 19.4.

First, let us assume that the crash occurs just after the log record for the step:

write(B)

of transaction T_0 has been written to stable storage (Figure 19.4a). When the system comes back up, it finds the record <T_0 start> in the log, but no corresponding <T_0 commit> or <T_0 abort> record. Thus, transaction T_0 must be undone, so an undo(T_0) is performed. As a result, the values in accounts A and B (on the disk) are restored to \$1000 and \$2000, respectively.

Next, let us assume that the crash comes just after the log record for the step:

write(C)

of transaction T_1 has been written to stable storage (Figure 19.4b). When the system comes back up, two recovery actions need to be taken. The operation undo(T_1) must be performed, since the record <T_1 start> appears in the log, but there is no record <T_1 commit> or <T_1 abort>. The operation redo(T_0) must be performed, since the log contains both the record <T_0 start> and the record <T commit>. At the end of

Telegram: @uni_k

www.konkur.in

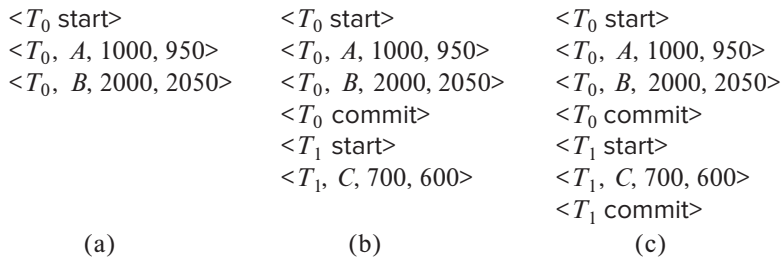


Figure 19.4 The same log, shown at three different times.

the entire recovery procedure, the values of accounts A, B, and C are \$950, \$2050, and \$700, respectively.

Finally, let us assume that the crash occurs just after the log record:

$$<T_1 \text{ commit}>$$

has been written to stable storage (Figure 19.4c). When the system comes back up, both T_0 and T_1 need to be redone, since the records $<T_0$ start> and $<T_0$ commit> appear in the log, as do the records $<T_1$ start> and $<T_1$ commit>. After the system performs the recovery procedures redo(T_0) and redo(T_1), the values in accounts A, B, and C are \$950, \$2050, and \$600, respectively.

19.3.6 Checkpoints

When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce **checkpoints**.

We describe below a simple checkpoint scheme that (a) does not permit any updates to be performed while the checkpoint operation is in progress, and (b) outputs all modified buffer blocks to disk when the checkpoint is performed. We discuss later how to modify the checkpointing and recovery procedures to provide more flexibility by relaxing both these requirements.

A checkpoint is performed as follows:

Telegram: @uni_k

www.konkur.in

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. We discuss how this requirement can be enforced in Section 19.5.2.

The presence of a <checkpoint L> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that completed prior to the checkpoint. For such a transaction, the <T_i commit> record (or < T_i abort> record) appears in the log before the <checkpoint> record. Any database modifications made by T_i must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on T_i.

After a system crash has occurred, the system examines the log to find the last <checkpoint L> record (this can be done by searching the log backward, from the end of the log, until the first <checkpoint L> record is found).

The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the <checkpoint L> record was written to the log. Let us denote this set of transactions as T.

- For all transactions T_k in T that have no <T_k commit> record or <T_k abort> record in the log, execute undo(T_k).
- For all transactions T_k in T such that either the record <T_k commit> or the record <T_k abort> appears in the log, execute redo(T_k).

Note that we need only examine the part of the log starting with the last checkpoint log record to find the set of transactions T and to find out whether a commit or abort record occurs in the log for each transaction in T.

As an illustration, consider the set of transactions $\{T_0, T_1, \ldots, T_{100}\}$. Suppose that the most recent checkpoint took place during the execution of transaction T_{67} and T_{69}, while T_{68} and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T_{67}, $T_{69}, \ldots, T_{100}$ need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (i.e., either committed or aborted); otherwise, it was incomplete and needs to be undone.

Consider the set of transactions L in a checkpoint log record. For each transaction T_i in L, log records of the transaction that occur prior to the checkpoint log record may be needed to undo the transaction, in case it does not commit. However, all log records prior to the earliest of the <T_i start> log records, among transactions T in L,

Telegram: @uni_k

www.konkur.in

are not needed once the checkpoint has completed. These log records can be erased whenever the database system needs to reclaim the space occupied by these records.

The requirement that transactions must not perform any updates to buffer blocks or to the log during checkpointing can be bothersome, since transaction processing has to halt while a checkpoint is in progress. A **fuzzy checkpoint** is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out. Section 19.5.4 describes fuzzy-checkpointing schemes. Later in Section 19.9 we describe a checkpoint scheme that is not only fuzzy, but does not even require all modified buffer blocks to be output to disk at the time of the checkpoint.

19.4 Recovery Algorithm

Until now, in discussing recovery, we have identified transactions that need to be redone and those that need to be undone, but we have not given a precise algorithm for performing these actions. We are now ready to present the full **recovery algorithm** using log records for recovery from transaction failure and a combination of the most recent checkpoint and log records to recover from a system crash.

The recovery algorithm described in this section requires that a data item that has been updated by an uncommitted transaction cannot be modified by any other transaction, until the first transaction has either committed or aborted. Recall that this restriction was discussed in Section 19.3.3.

19.4.1 Transaction Rollback

First consider transaction rollback during normal operation (i.e., not during recovery from a system crash). Rollback of a transaction T_i is performed as follows:

1. The log is scanned backward, and for each log record of T_i of the form $<T_i, X_j, V_1, V_2>$ that is found:
   a. The value V_1 is written to data item X_j, and
   b. A special redo-only log record $<T_i, X_j, V_1>$ is written to the log, where V_1 is the value being restored to data item X_j during the rollback. These log records are sometimes called **compensation log records**. Such records do not need undo information, since we never need to undo such an undo operation. We shall explain later how they are used.
2. Once the log record $<T_i$ start$>$ is found, the backward scan is stopped, and a log record $<T_i$ abort$>$ is written to the log.

Observe that every update action performed by the transaction or on behalf of the transaction, including actions taken to restore data items to their old value, have now been recorded in the log. In Section 19.4.2 we shall see why this is a good idea.

Telegram: @uni_k

www.konkur.in

19.4.2 Recovery After a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1. In the **redo phase**, the system replays updates of *all* transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred.

   This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back. Such incomplete transactions would either have been active at the time of the checkpoint, and thus would appear in the transaction list in the checkpoint record, or would have started later; further, such incomplete transactions would have neither a $<T_i$ abort> nor a $<T_i$ commit> record in the log.

   The specific steps taken while scanning the log are as follows:

   a. The list of transactions to be rolled back, undo-list, is initially set to the list L in the <checkpoint L> log record.

   b. Whenever a normal log record of the form $<T_i, X_j, V_1, V_2>$, or a redo-only log record of the form $<T_i, X_j, V_2>$ is encountered, the operation is redone; that is, the value V_2 is written to data item X_j.

   c. Whenever a log record of the form $<T_i$ start> is found, T_i is added to undo-list.

   d. Whenever a log record of the form $<T_i$ abort> or $<T_i$ commit> is found, T_i is removed from undo-list.

   At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

2. In the **undo phase**, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

   a. Whenever it finds a log record belonging to a transaction in the undo-list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

   b. When the system finds a $<T_i$ start> log record for a transaction T_i in undo-list, it writes a $<T_i$ abort> log record to the log and removes T_i from undo-list.

   c. The undo phase terminates once undo-list becomes empty, that is, the system has found $<T_i$ start> log records for all transactions that were initially in undo-list.

Telegram: @uni_k

www.konkur.in

After the undo phase of recovery terminates, normal transaction processing can resume.

Observe that the redo phase replays every log record since the most recent checkpoint record. In other words, this phase of restart recovery repeats all the update actions that were executed after the checkpoint, and whose log records reached the stable log. The actions include actions of incomplete transactions and the actions carried out to roll back failed transactions. The actions are repeated in the same order in which they were originally carried out; hence, this process is called **repeating history**. Although it may appear wasteful, repeating history even for failed transactions simplifies recovery schemes.

Figure 19.5 shows an example of actions logged during normal operation and actions performed during failure recovery. In the log shown in the figure, transaction T_1 had committed, and transaction T_0 had been completely rolled back, before the system crashed. Observe how the value of data item B is restored during the rollback of T_0. Observe also the checkpoint record, with the list of active transactions containing T_0 and T_1.

When recovering from a crash, in the redo phase, the system performs a redo of all operations after the last checkpoint record. In this phase, the list undo-list initially contains T_0 and T_1; T_1 is removed first when its commit log record is found, while T_2 is added when its start log record is found. Transaction T_0 is removed from undo-list when its abort log record is found, leaving only T_2 in undo-list. The undo phase scans the log backwards from the end, and when it finds a log record of T_2 updating A, the old value of A is restored, and a redo-only log record is written to the log. When the

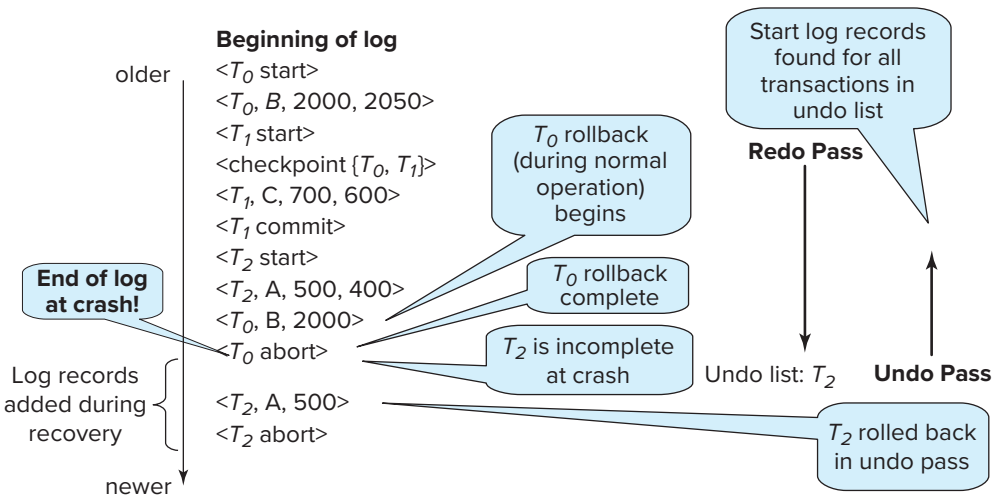


Figure 19.5 Example of logged actions and actions during recovery.

Telegram: @uni_k

www.konkur.in

start record for T_2 is found, an abort record is added for T_2. Since undo-list contains no more transactions, the undo phase terminates, completing recovery.

19.4.3 Optimizing Commit Processing

Committing a transaction requires that its log records have been forced to disk. If a separate log flush is done for each transaction, each commit incurs a significant log write overhead. The rate of transaction commit can be increased using the **group-commit** technique. With this technique, instead of attempting to force the log as soon as a transaction completes, the system waits until several transactions have completed, or a certain period of time has passed since a transaction completed execution. It then commits the group of transactions that are waiting, together. Blocks written to the log on stable storage would contain records of several transactions. By careful choice of group size and maximum waiting time, the system can ensure that blocks are full when they are written to stable storage without making transactions wait excessively. This technique results, on average, in fewer output operations per committed transaction.

If logging is done to hard disk, writing a block of data can take about 5 to 10 milliseconds. As a result, without group commit, at most 100 to 200 transactions can be committed per second. If records of 10 transactions fit in a disk block, group commit will allow 1000 to 2000 transactions to be committed per second.

If logging is done to flash, writing a block can take about 100 microseconds, allowing 10,000 transactions to be committed per second without group commit. If records of 10 transactions fit in a disk block, group commit will allow 100,000 transactions to be committed per second on flash. A further benefit of group commit with flash is that it minimizes the number of times the same page is written, which in turn minimizes the number of erase operations, which can be expensive. (Recall that flash storage systems remap logical pages to a pre-erased physical page, avoiding delay at the time a page is written, but the erase operation must be performed eventually as part of garbage collection of old versions of pages.)

Although group commit reduces the overhead imposed by logging, it results in a slight delay in commit of transactions that perform updates. When the rate of commits is low, the delay may not be worth the benefit, but with high rates of transaction commit, the overall delay in commit is actually reduced by using group commit.

In addition to optimizations done at the database, programmers can also take some steps to improve transaction commit performance. For example, consider an application that loads data into a database. If the application performs each insert as a separate transaction, the number of inserts that can be performed per second is limited by the number of blocks writes that can be performed per second. If the application waits for one insert to finish before starting the next one, group commit does not offer any benefits and in fact may slow the system down. However, in such a case, performance can be significantly improved by performing a batch of inserts as a single transaction. The log records corresponding to multiple inserts are then written together in one page. The number of inserts that can be performed per second then increases correspondingly.

Telegram: @uni_k

www.konkur.in

19.5 Buffer Management

In this section, we consider several subtle details that are essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

19.5.1 Log-Record Buffering

So far, we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level. Furthermore, as we saw in Section 19.2.1, the output of a block to stable storage may involve several output operations at the physical level.

The cost of outputting a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer and output to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

- Transaction T_i enters the commit state after the $<T_i$ commit> log record has been output to stable storage.
- Before the $<T_i$ commit> log record can be output to stable storage, all log records pertaining to transaction T_i must have been output to stable storage.
- Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.

  This rule is called the **write-ahead logging (WAL)** rule. (Strictly speaking, the WAL rule requires only that the undo information in the log has been output to stable storage, and it permits the redo information to be written later. The difference is relevant in systems where undo information and redo information are stored in separate log records.)

The three rules state situations in which certain log records *must* have been output to stable storage. There is no problem resulting from the output of log records *earlier* than necessary. Thus, when the system finds it necessary to output a log record to

Telegram: @uni_k

www.konkur.in

stable storage, it outputs an entire block of log records, if there are enough log records in main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block and are output to stable storage.

Writing the buffered log to disk is sometimes referred to as a **log force**.

19.5.2 Database Buffering

In Section 19.2.2, we described the use of a two-level storage hierarchy. The system stores the database in non-volatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block B_1 in main memory when another block B_2 needs to be brought into memory. If B_1 has been modified, B_1 must be output prior to the input of B_2. As discussed in Section 13.5.1 this storage hierarchy is similar to the standard operating-system concept of *virtual memory*.

One might expect that transactions would force-output all modified blocks to disk when they commit. Such a policy is called the **force** policy. The alternative, the **no-force** policy, allows a transaction to commit even if it has modified some blocks that have not yet been written back to disk. All the recovery algorithms described in this chapter work correctly even with the no-force policy. The no-force policy allows faster commit of transactions; moreover it allows multiple updates to accumulate on a block before it is output to stable storage, which can reduce the number of output operations greatly for frequently updated blocks. As a result, the standard approach taken by most systems is the no-force policy.

Similarly, one might expect that blocks modified by a transaction that is still active should not be written to disk. This policy is called the **no-steal** policy. The alternative, the **steal** policy, allows the system to write modified blocks to disk even if the transactions that made those modifications have not all committed. As long as the write-ahead logging rule is followed, all the recovery algorithms we study in the chapter work correctly even with the steal policy. Further, the no-steal policy does not work with transactions that perform a large number of updates, since the buffer may get filled with updated pages that cannot be evicted to disk, and the transaction cannot then proceed. As a result, the standard approach taken by most systems is the steal policy.

To illustrate the need for the write-ahead logging requirement, consider our banking example with transactions T_0 and T_1. Suppose that the state of the log is:

$$<T_0 \text{ start}>$$
$$<T_0, A, 1000, 950>$$

and that transaction T_0 issues a read(B). Assume that the block on which B resides is not in main memory and that main memory is full. Suppose that the block on which A resides is chosen to be output to disk. If the system outputs this block to disk and then a crash occurs, the values in the database for accounts A, B, and C are \$950, \$2000, and

Telegram: @uni_k

www.konkur.in

$700, respectively. This database state is inconsistent. However, because of the WAL requirements, the log record:

$$<T_0, A, 1000, 950>$$

must be output to stable storage prior to output of the block on which A resides. The system can use the log record during recovery to bring the database back to a consistent state.

When a block B_1 is to be output to disk, all log records pertaining to data in B_1 must be output to stable storage before B_1 is output. It is important that no writes to the block B_1 be in progress while the block is being output, since such a write could violate the write-ahead logging rule. We can ensure that there are no writes in progress by using a special means of locking:

- Before a transaction performs a write on a data item, it acquires an exclusive lock on the block in which the data item resides. The lock is released immediately after the update has been performed.
- The following sequence of actions is taken when a block is to be output:
  - Obtain an exclusive lock on the block, to ensure that no transaction is performing a write on the block.
  - Output log records to stable storage until all log records pertaining to block B_1 have been output.
  - Output block B_1 to disk.
  - Release the lock once the block output has completed.

Locks on buffer blocks are unrelated to locks used for concurrency control of transactions, and releasing them in a non-two-phase manner does not have any implications on transaction serializability. These locks, and other similar locks that are held for a short duration, are often referred to as **latches**.

Locks on buffer blocks can also be used to ensure that buffer blocks are not updated, and log records are not generated, while a checkpoint is in progress. This restriction may be enforced by acquiring exclusive locks on all buffer blocks, as well as an exclusive lock on the log, before the checkpoint operation is performed. These locks can be released as soon as the checkpoint operation has completed.

Database systems usually have a process that continually cycles through the buffer blocks, outputting modified buffer blocks back to disk. The above locking protocol must of course be followed when the blocks are output. As a result of continuous output of modified blocks, the number of **dirty blocks** in the buffer, that is, blocks that have been modified in the buffer but have not been subsequently output, is minimized. Thus, the number of blocks that have to be output during a checkpoint is minimized; further,

Telegram: @uni_k

www.konkur.in

when a block needs to be evicted from the buffer, it is likely that there will be a non-dirty block available for eviction, allowing the input to proceed immediately instead of waiting for an output to complete.

19.5.3 Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements in Section 19.5.2.

   This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, non-database applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements in Section 19.5.1, the operating system should not write out the database buffer pages itself, but instead should request the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage.

   Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called **swap space**. If the operating system decides to output a block B_x, that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks.

   Therefore, if the database buffer is in virtual memory, transfers between database files and the buffer in virtual memory must be managed by the database system, which enforces the write-ahead logging requirements that we discussed.

   This approach may result in extra output of data to disk. If a block B_x is output by the operating system, that block is not output to the database. Instead, it is output to the swap space for the operating system's virtual memory. When the database system needs to output B_x, the operating system may need first to input B_x from its swap space. Thus, instead of a single output of B_x, there may be two outputs of B_x (one by the operating system, and one by the database system) and one extra input of B_x.

Telegram: @uni_k

www.konkur.in

Although both approaches suffer from some drawbacks, one or the other must be chosen unless the operating system is designed to support the requirements of database logging.

19.5.4 Fuzzy Checkpointing

The checkpointing technique described in Section 19.3.6 requires that all updates to the database be temporarily suspended while the checkpoint is in progress. If the number of pages in the buffer is large, a checkpoint may take a long time to finish, which can result in an unacceptable interruption in processing of transactions.

To avoid such interruptions, the checkpointing technique can be modified to permit updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk. The checkpoint thus generated is a **fuzzy checkpoint**.

Since pages are output to disk only after the checkpoint record has been written, it is possible that the system could crash before all pages are written. Thus, a checkpoint on disk may be incomplete. One way to deal with incomplete checkpoints is this: The location in the log of the checkpoint record of the last completed checkpoint is stored in a fixed position, last-checkpoint, on disk. The system does not update this information when it writes the checkpoint record. Instead, before it writes the checkpoint record, it creates a list of all modified buffer blocks. The last-checkpoint information is updated only after all buffer blocks in the list of modified buffer blocks have been output to disk.

Even with fuzzy checkpointing, a buffer block must not be updated while it is being output to disk, although other buffer blocks may be updated concurrently. The write-ahead log protocol must be followed so that (undo) log records pertaining to a block are on stable storage before the block is output.

19.6 Failure with Loss of Non-Volatile Storage

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the non-volatile storage remains intact. Although failures in which the content of non-volatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure. In this section, we discuss only disk storage. Our discussions apply as well to other non-volatile storage types.

The basic scheme is to **dump** the entire contents of the database to stable storage periodically—say, once per day. For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

One approach to database dumping requires that no transaction may be active during the dump procedure, and it uses a procedure similar to checkpointing:

Telegram: @uni_k

www.konkur.in

1. Output all log records currently residing in main memory onto stable storage.
2. Output all buffer blocks onto the disk.
3. Copy the contents of the database to stable storage.
4. Output a log record <dump> onto the stable storage.

Steps 1, 2, and 4 correspond to the three steps used for checkpoints in Section 19.3.6.

To recover from the loss of non-volatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the actions since the most recent dump occurred. Notice that no undo operations need to be executed.

In case of a partial failure of non-volatile storage, such as the failure of a single block or a few blocks, only those blocks need to be restored, and redo actions performed only for those blocks.

A dump of the database contents is also referred to as an **archival dump**, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and checkpointing of buffers are similar.

Most database systems also support an **SQL dump**, which writes out SQL DDL statements and SQL insert statements to a file, which can then be reexecuted to re-create the database. Such dumps are useful when migrating data to a different instance of the database, or to a different version of the database software, since the physical locations and layout may be different in the other database instance or database software version.

The simple dump procedure described here is costly for the following two reasons. First, the entire database must be copied to stable storage, resulting in considerable data transfer. Second, since transaction processing is halted during the dump procedure, CPU cycles are wasted. **Fuzzy dump** schemes have been developed that allow transactions to be active while the dump is in progress. They are similar to fuzzy-checkpointing schemes; see the bibliographical notes for more details.

19.7 High Availability Using Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Today's applications need transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**; that is, the time for which the system is unusable must be extremely short.

We can achieve high availability by performing transaction processing at one site, called the **primary site**, and having a **remote backup** site where all the data from the primary site are replicated. The remote backup site is sometimes also called the **secondary site**. The remote site must be kept synchronized with the primary site as updates are performed at the primary. We achieve synchronization by sending all log records from the primary site to the remote backup site. The remote backup site must be physically

Telegram: @uni_k

www.konkur.in

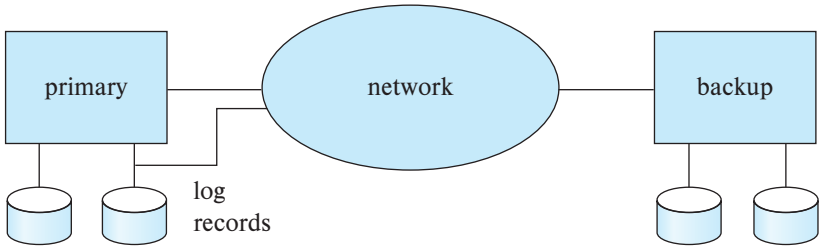


Figure 19.6 Architecture of remote backup system.

separated from the primary—for example, we can locate it in a different state—so that a disaster such as a fire, flood or an earthquake at the primary does not also damage the remote backup site.[3] Figure 19.6 shows the architecture of a remote backup system.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost.

Several issues must be addressed in designing a remote backup system:

- **Detection of failure**. It is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup. For example, several independent network connections, including perhaps a modem connection over a telephone line, may be used. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.
- **Transfer of control**. When the primary fails, the backup site takes over processing and becomes the new primary. The decision to transfer control can be done manually or can be automated using software provided by database system vendors.

  Queries must now be sent to the new primary. To do so automatically, many systems assign the IP address of the old primary to the new primary. Existing database connections will fail, but when an application tries to reopen a connection it gets connected to the new primary. Some systems instead use a **high availability proxy**

[3]Since earthquakes can cause damage over a wide area, the backup is generally required to be in a different seismic zone.

Telegram: @uni_k

www.konkur.in

machine. Application clients do not connect to the database directly, but connect through the proxy. The proxy transparently routes application requests to the current primary. (There can be more than one machine acting as proxy at the same time, to deal with a situation where a proxy machine fails; requests can be routed through any active proxy machine.)

When the original primary site recovers, it can either play the role of remote backup or it can take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down. The old primary must catch up with the updates in the log by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the new primary (which is the old backup site) can pretend to have failed, resulting in the old primary taking over.

- **Time to recover**. If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received and can perform a checkpoint so that earlier parts of the log can be deleted. The delay before the remote backup takes over can be significantly reduced as a result.

  A **hot-spare configuration** can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

- **Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows:

  - **One-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary site.

    The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site. Thus, human intervention may be required to bring the database to a consistent state.

  - **Two-very-safe**. A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.

    The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.

Telegram: @uni_k

www.konkur.in

- **Two-safe**. This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

  This scheme provides better availability than does two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost.

Most database systems today provide support for replication to a backup copy, along with support for hot spares and quick switchover from the primary to the backup. Many database systems also allow replication to more than one backup; such a feature can be used to provide a local backup to deal with machine failures, along with a remote backup to deal with disasters.

Although update transactions cannot be executed at a backup server, many database systems allow read-only queries to be executed at backup servers. The load at the primary can be reduced by executing at least some of the read-only transactions at the backup. Snapshot-isolation can be used at the backup server to give readers a transaction consistent view of the data, while ensuring that updates are never blocked from being applied at the backup.

Remote backup is also supported at the level of file systems, typically by network file system or NAS implementations, as well as at the disk level, typically by storage area network (SAN) implementations. Remote backups are kept synchronized with the primary by ensuring that all block writes performed at the primary are also replicated at the backup. File-system level and disk level backups can be used to replicate the database data as well as log files. If the primary fails, the backup system can recover using its replica of the data and log files. However, to ensure that recovery will work correctly at the backup site, the file system level replication must be done in a way that ensures that the write-ahead logging (WAL) rule continues to hold. To do so, if the database forces a block to disk and then performs some other update actions at the primary, the block must also be forced to disk at the backup, before subsequent updates are performed at the backup system.

An alternative way of achieving high availability is to use a *distributed database*, with data replicated at more than one site. Transactions are then required to update all replicas of any data item that they update. We study distributed databases, including replication, in Chapter 23. When properly implemented, distributed databases can provide a higher level of availability than remote backup systems, but are more complex and expensive to implement and maintain.

End-users typically interact with applications, rather than directly with database. To ensure availability of an application, as well as to support handling of a large number of requests per second, applications may run on multiple application servers. Requests from clients are **load-balanced** across the servers. The load-balancer ensures that all requests from a particular client are sent to a single application server, as long as the

Telegram: @uni_k

www.konkur.in

application server is functional. If an application server fails, client requests are routed to other application servers, so users can continue to use the application. Although users may notice a small interruption, application servers can ensure that a user is not forced to login again, by sharing session information across application servers.

19.8 Early Lock Release and Logical Undo Operations

Any index used in processing a transaction, such as a B^+-tree, can be treated as normal data, but to increase concurrency, we can use the B^+-tree concurrency-control algorithm described in Section 18.10.2 to allow locks to be released early, in a non-two-phase manner. As a result of early lock release, it is possible that a value in a B^+-tree node is updated by one transaction T_1, which inserts an entry $(V1, R1)$, and subsequently by another transaction T_2, which inserts an entry $(V2, R2)$ in the same node, moving the entry $(V1, R1)$ even before T_1 completes execution.[4] At this point, we cannot undo transaction T_1 by replacing the contents of the node with the old value prior to T_1 performing its insert, since that would also undo the insert performed by T_2; transaction T_2 may still commit (or may have already committed). In this example, the only way to undo the effect of insertion of $(V1, R1)$ is to execute a corresponding delete operation.

In the rest of this section, we see how to extend the recovery algorithm of Section 19.4 to support early lock release.

19.8.1 Logical Operations

The insertion and deletion operations are examples of a class of operations that require logical undo operations since they release locks early; we call such operations **logical operations**. Such early lock release is important not only for indices, but also for operations on other system data structures that are accessed and updated very frequently; examples include data structures that track the blocks containing records of a relation, the free space in a block, and the free blocks in a database. If locks were not released early after performing operations on such data structures, transactions would tend to run serially, affecting system performance.

The theory of conflict serializability has been extended to operations, based on what operations conflict with what other operations. For example, two insert operations on a B^+-tree do not conflict if they insert different key values, even if they both update overlapping areas of the same index page. However, insert and delete operations conflict with other insert and delete operations, as well as with read operations, if they use the same key value. See the bibliographical notes for references to more information on this topic.

Operations acquire *lower-level locks* while they execute but release them when they complete; the corresponding transaction must however retain a *higher-level lock* in a

[4]Recall that an entry consists of a key value and a record identifier, or a key value and a record in the case of the leaf level of a B^+-tree file organization.

Telegram: @uni_k

www.konkur.in

two-phase manner to prevent concurrent transactions from executing conflicting actions. For example, while an insert operation is being performed on a B$^+$-tree page, a short-term lock is obtained on the page, allowing entries in the page to be shifted during the insert; the short-term lock is released as soon as the page has been updated. Such early lock release allows a second insert to execute on the same page. However, each transaction must obtain a lock on the key values being inserted or deleted and retain it in a two-phase manner, to prevent a concurrent transaction from executing a conflicting read, insert, or delete operation on the same key value.

Once the lower-level lock is released, the operation cannot be undone by using the old values of updated data items and must instead be undone by executing a compensating operation; such an operation is called a **logical undo operation**. It is important that the lower-level locks acquired during an operation are sufficient to perform a subsequent logical undo of the operation, for reasons explained later in Section 19.8.4.

19.8.2 Logical Undo Log Records

To allow logical undo of operations, before an operation is performed to modify an index, the transaction creates a log record $<T_i, O_j,$ operation-begin$>$, where O_j is a unique identifier for the operation instance.[5] While the system is executing the operation, it creates update log records in the normal fashion for all updates performed by the operation. Thus, the usual old-value and new-value information is written out as usual for each update performed by the operation; the old-value information is required in case the transaction needs to be rolled back before the operation completes. When the operation finishes, it writes an operation-end log record of the form $<T_i, O_j,$ operation-end, $U>$, where the U denotes undo information.

For example, if the operation inserted an entry in a B$^+$-tree, the undo information U would indicate that a deletion operation is to be performed and would identify the B$^+$-tree and what entry to delete from the tree. Such logging of information about operations is called **logical logging**. In contrast, logging of old-value and new-value information is called **physical logging**, and the corresponding log records are called **physical log records**.

Note that in the above scheme, logical logging is used only for undo, not for redo; redo operations are performed exclusively using physical log record. This is because the state of the database after a system failure may reflect some updates of an operation and not other operations, depending on what buffer blocks had been written to disk before the failure. Data structures such as B$^+$-trees would not be in a consistent state, and neither logical redo nor logical undo operations can be performed on an inconsistent data structure. To perform logical redo or undo, the database state on disk must be **operation consistent**, that is, it should not have partial effects of any operation. However, as we shall see, the physical redo processing in the redo phase of the recovery scheme, along with undo processing using physical log records, ensures that the parts of the

[5]The position in the log of the operation-begin log record can be used as

Telegram: @uni_k

www.konkur.in

database accessed by a logical undo operation are in an operation consistent state before the logical undo operation is performed.

An operation is said to be **idempotent** if executing it several times in a row gives the same result as executing it once. Operations such as inserting an entry into a B^+-tree may not be idempotent, and the recovery algorithm must therefore make sure that an operation that has already been performed is not performed again. On the other hand, a physical log record is idempotent, since the corresponding data item would have the same value regardless of whether the logged update is executed one or multiple times.

19.8.3 Transaction Rollback with Logical Undo

When rolling back a transaction T_i, the log is scanned backwards, and log records corresponding to T_i are processed as follows:

1. Physical log records encountered during the scan are handled as described earlier, except those that are skipped as described shortly. Incomplete logical operations are undone using the physical log records generated by the operation.
2. Completed logical operations, identified by operation-end records, are rolled back differently. Whenever the system finds a log record $<T_i, O_j$, operation-end, $U>$, it takes special actions:

   a. It rolls back the operation by using the undo information U in the log record. It logs the updates performed during the rollback of the operation just like updates performed when the operation was first executed.

      At the end of the operation rollback, instead of generating a log record $<T_i, O_j$, operation-end, $U>$, the database system generates a log record $<T_i, O_j$, operation-abort$>$.

   b. As the backward scan of the log continues, the system skips all log records of transaction T_i until it finds the log record $<T_i, O_j$, operation-begin$>$. After it finds the operation-begin log record, it processes log records of transaction T_i in the normal manner again.

   Observe that the system logs physical undo information for the updates performed during rollback, instead of using redo-only compensation log records. This is because a crash may occur while a logical undo is in progress, and on recovery the system has to complete the logical undo; to do so, restart recovery will undo the partial effects of the earlier undo, using the physical undo information, and then perform the logical undo again.

   Observe also that skipping over physical log records when the operation-end log record is found during rollback ensures that the old values in the physical log record are not used for rollback once the operation completes.

Telegram: @uni_k

www.konkur.in

3. If the system finds a record $<T_i, O_j,$ operation-abort$>$, it skips all preceding records (including the operation-end record for O_j) until it finds the record $<T_i, O_j,$ operation-begin$>$.

   An operation-abort log record would be found only if a transaction that is being rolled back had been partially rolled back earlier. Recall that logical operations may not be idempotent, and hence a logical undo operation must not be performed multiple times. These preceding log records must be skipped to prevent multiple rollback of the same operation in case there had been a crash during an earlier rollback and the transaction had already been partly rolled back.

4. As before, when the $<T_i$ start$>$ log record has been found, the transaction rollback is complete, and the system adds a record $<T_i$ abort$>$ to the log.

If a failure occurs while a logical operation is in progress, the operation-end log record for the operation will not be found when the transaction is rolled back. However, for every update performed by the operation, undo information—in the form of the old value in the physical log records—is available in the log. The physical log records will be used to roll back the incomplete operation.

Now suppose an operation undo was in progress when the system crash occurred, which could happen if a transaction was being rolled back when the crash occurred.

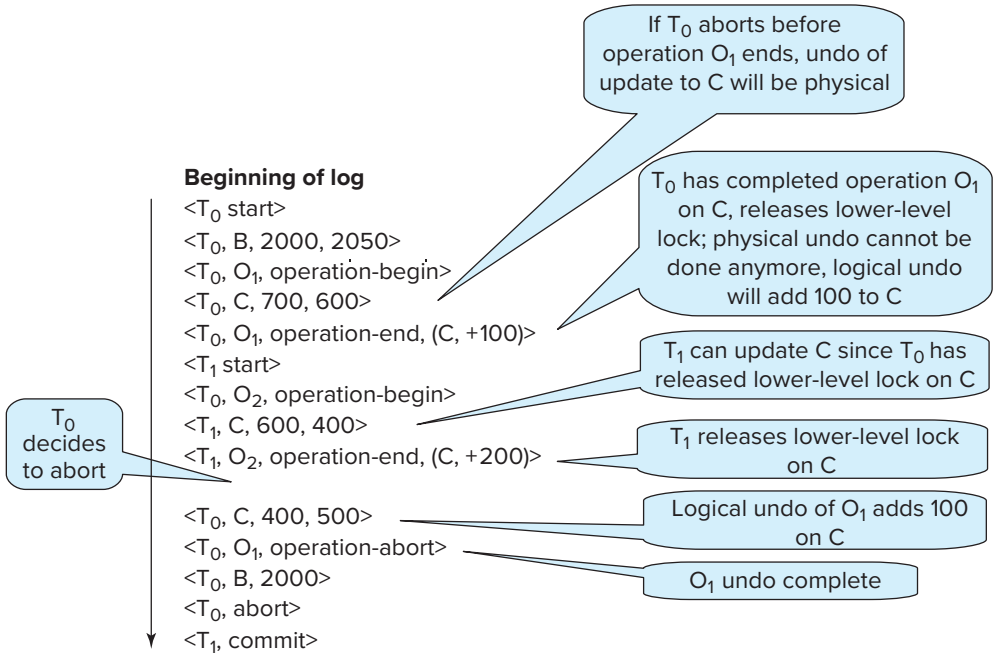


Figure 19.7 Transaction rollback with logical undo operations.

Telegram: @uni_k

www.konkur.in

Then the physical log records written during operation undo would be found, and the partial operation undo would itself be undone using these physical log records. Continuing in the backward scan of the log, the original operation's operation-end record would then be found, and the operation undo would be executed again. Rolling back the partial effects of the earlier undo operation using the physical log records brings the database to a consistent state, allowing the logical undo operation to be executed again.

Figure 19.7 shows an example of a log generated by two transactions, which add or subtract a value from a data item. Early lock release on the data item C by transaction T_0 after operation O_1 completes allows transaction T_1 to update the data item using O_2, even before T_0 completes, but necessitates logical undo. The logical undo operation needs to add or subtract a value from the data item instead of restoring an old value to the data item.

The annotations on the figure indicate that before an operation completes, rollback can perform physical undo; after the operation completes and releases lower-level locks, the undo must be performed by subtracting or adding a value, instead of restoring the old value. In the example in the figure, T_0 rolls back operation O_1 by adding 100 to C; on the other hand, for data item B, which was not subject to early lock release, undo is performed physically. Observe that T_1, which had performed an update on C, commits, and its update O_2, which added 200 to C and was performed before the undo of O_1, has persisted even though O_1 has been undone.

Figure 19.8 shows an example of recovery from a crash with logical undo logging. In this example, operation T_1 was active and executing operation O_4 at the time of checkpoint. In the redo pass, the actions of O_4 that are after the checkpoint log record are redone. At the time of crash, operation O_5 was being executed by T_2, but the operation was not complete. The undo-list contains T_1 and T_2 at the end of the redo pass. During the undo pass, the undo of operation O_5 is carried out using the old value in the physical log record, setting C to 400; this operation is logged using a redo-only log record. The start record of T_2 is encountered next, resulting in the addition of $<T_2$ abort> to the log and removal of T_2 from undo-list.

The next log record encountered is the operation-end record of O_4; logical undo is performed for this operation by adding 300 to C, which is logged physically, and an operation-abort log record is added for O_4. The physical log records that were part of O_4 are skipped until the operation-begin log record for O_4 is encountered. In this example, there are no other intervening log records, but in general log records from other transactions may be found before we reach the operation-begin log record; such log records should of course not be skipped (unless they are part of a completed operation for the corresponding transaction and the algorithm skips those records). After the operation-begin log record is found for O_4, a physical log record is found for T_1, which is rolled back physically. Finally the start log record for T_1 is found; this results in $< T_1$ abort> being added to the log and T_1 being deleted from undo-list. At this point undo-list is empty, and the undo phase is complete.

Telegram: @uni_k

www.konkur.in

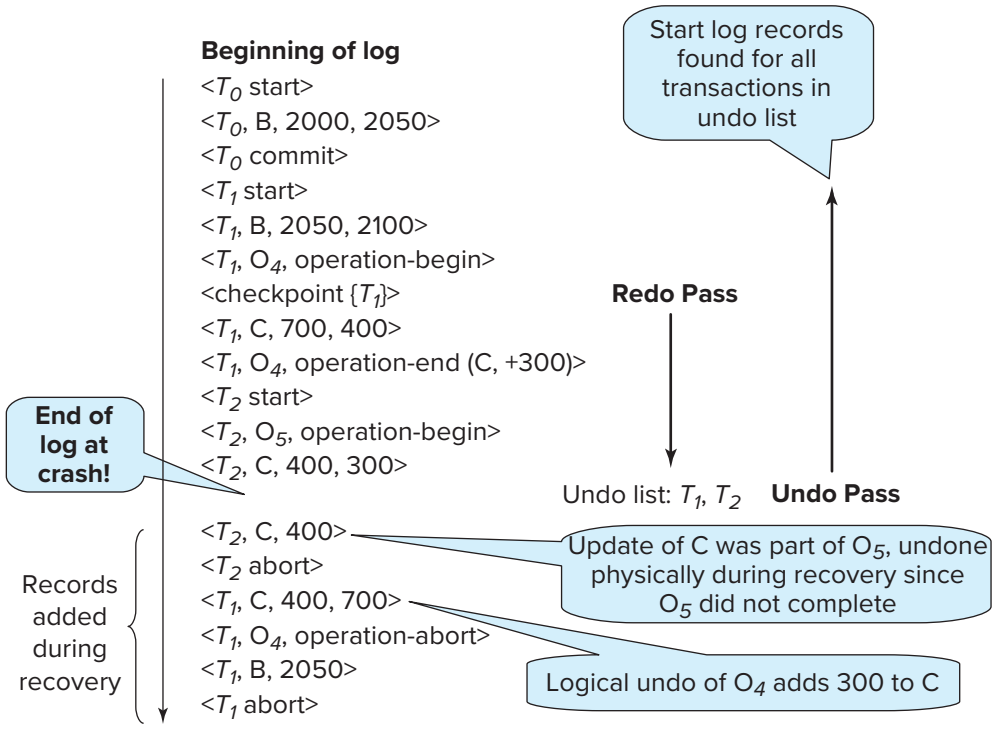


Figure 19.8 Failure recovery actions with logical undo operations.

19.8.4 Concurrency Issues in Logical Undo

As mentioned earlier, it is important that the lower-level locks acquired during an operation are sufficient to perform a subsequent logical undo of the operation; otherwise concurrent operations that execute during normal processing may cause problems in the undo phase. For example, suppose the logical undo of operation O_1 of transaction T_1 can conflict at the data item level with a concurrent operation O_2 of transaction T_2, and O_1 completes while O_2 does not. Assume also that neither transaction had committed when the system crashed. The physical update log records of O_2 may appear before and after the operation-end record for O_1, and during recovery updates done during the logical undo of O_1 may get fully or partially overwritten by old values during the physical undo of O_2. This problem cannot occur if O_1 had obtained all the lower-level locks required for the logical undo of O_1, since then there cannot be such a concurrent O_2.

If both the original operation and its logical undo operation access a single page (such operations are called physiological operations and are discussed in Section 19.9), the locking requirement above is met easily. Otherwise the details of the specific operation need to be considered when deciding on what lower-level locks need to be obtained. For example, update operations on a B^+-tree could obtain a short-term lock on the root,

Telegram: @uni_k

www.konkur.in

to ensure that operations execute serially. See the bibliographical notes for references on B^+-tree concurrency control and recovery exploiting logical undo logging. See the bibliographical notes also for references to an alternative approach, called multilevel recovery, which relaxes this locking requirement.

19.9 ARIES

The state of the art in recovery methods is best illustrated by the **ARIES** recovery method. The recovery technique that we described in Section 19.4, along with the logical undo logging techniques described in Section 19.8, are modeled after ARIES, but they have been simplified significantly to bring out key concepts and make them easier to understand. In contrast, ARIES uses a number of techniques to reduce the time taken for recovery and to reduce the overhead of checkpointing. In particular, ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged. The price paid is greater complexity; the benefits are worth the price.

The four major differences between ARIES and the recovery algorithm presented earlier are that ARIES:

1. Uses a **log sequence number** (**LSN**) to identify log records and stores LSNs in database pages to identify which operations have been applied to a database page.
2. Supports **physiological redo** operations, which are physical in that the affected page is physically identified but can be logical within the page.

   For instance, the deletion of a record from a page may result in many other records in the page being shifted, if a slotted page structure (Section 13.2.2) is used. With physical redo logging, all bytes of the page affected by the shifting of records must be logged. With physiological logging, the deletion operation can be logged, resulting in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.
3. Uses a **dirty page table** to minimize unnecessary redos during recovery. As mentioned earlier, dirty pages are those that have been updated in memory, and the disk version is not up-to-date.
4. Uses a fuzzy-checkpointing scheme that records only information about dirty pages and associated information and does not even require writing of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

In the rest of this section, we provide an overview of ARIES. The bibliographical notes list some references that provide a complete description of ARIES.

Telegram: @uni_k

www.konkur.in

19.9.1 Data Structures

Each log record in ARIES has a log sequence number (LSN) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically, ARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file. The LSN then consists of a file number and an offset within the file.

Each page also maintains an identifier called the **PageLSN**. Whenever an update operation (whether physical or physiological) occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page. During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page. In combination with a scheme for recording PageLSNs as part of checkpointing, which we present later, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby, recovery time is reduced significantly.

The PageLSN is essential for ensuring idempotence in the presence of physiological redo operations, since reapplying a physiological redo that has already been applied to a page could cause incorrect changes to a page.

Pages should not be flushed to disk while an update is in progress, since physiological operations cannot be redone on the partially updated state of the page on disk. Therefore, ARIES uses latches on buffer pages to prevent them from being written to disk while they are being updated. It releases the buffer page latch only after the update is completed and the log record for the update has been written to the log.

Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log. There are special redo-only log records generated during transaction rollback, called **compensation log records** (**CLRs**) in ARIES. These serve the same purpose as the redo-only log records in our earlier recovery scheme. In addition, CLRs serve the role of the operation-abort log records in our scheme. The CLRs have an extra field, called the UndoNextLSN, that records the LSN of the log that needs to be undone next, when the transaction is being rolled back. This field serves the same purpose as the operation identifier in the operation-abort log record in our earlier recovery scheme, which helps to skip over log records that have already been rolled back.

The **DirtyPageTable** contains a list of pages that have been updated in the database buffer. For each page, it stores the PageLSN and a field called the RecLSN, which helps identify log records that have been applied already to the version of the page on disk. When a page is inserted into the DirtyPageTable (when it is first modified in the buffer pool), the value of RecLSN is set to the current end of log. Whenever the page is flushed to disk, the page is removed from the DirtyPageTable.

Telegram: @uni_k

www.konkur.in

A **checkpoint log record** contains the DirtyPageTable and a list of active transactions. For each transaction, the checkpoint log record also notes LastLSN, the LSN of the last log record written by the transaction. A fixed position on disk also notes the LSN of the last (complete) checkpoint log record.

Figure 19.9 illustrates some of the data structures used in ARIES. The log records shown in the figure are prefixed by their LSN; these may not be explicitly stored, but inferred from the position in the log, in an actual implementation. The data item identifier in a log record is shown in two parts, for example, 4894.1; the first identifies the page, and the second part identifies a record within the page (we assume a slotted page record organization within a page). Note that the log is shown with the newest records on top, since older log records, which are on disk, are shown lower in the figure.

Each page (whether in the buffer or on disk) has an associated PageLSN field. You can verify that the LSN for the last log record that updated page 4894 is 7567. By comparing PageLSNs for the pages in the buffer with the PageLSNs for the corresponding pages in stable storage, you can observe that the DirtyPageTable contains entries for all pages in the buffer that have been modified since they were fetched from stable storage. The RecLSN entry in the DirtyPageTable reflects the LSN at the end of the log

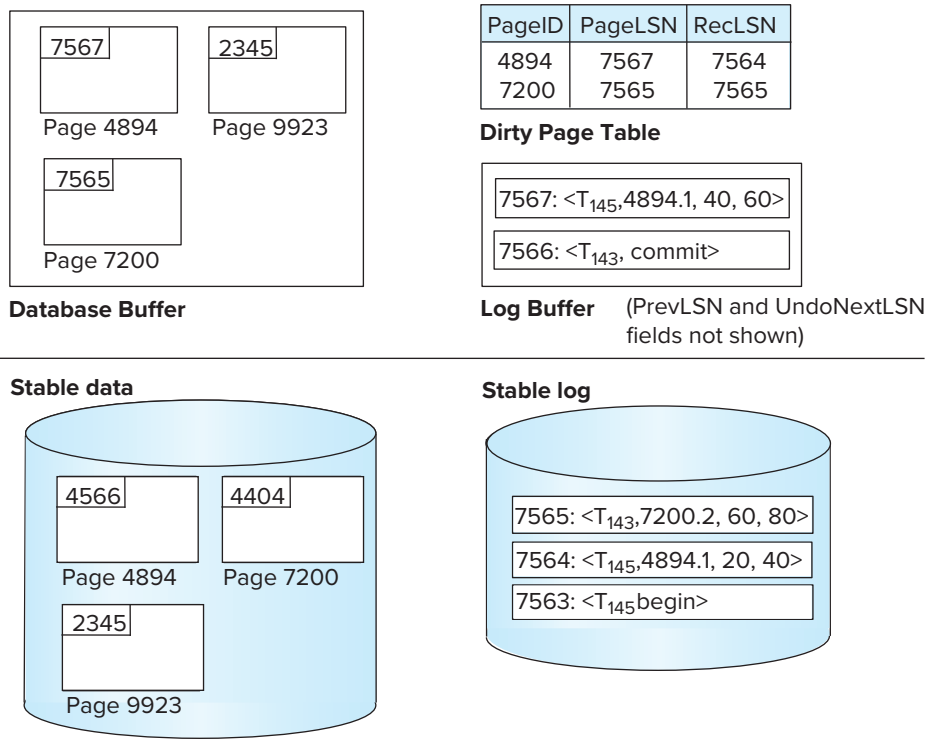


Figure 19.9 Data structures used in ARIES.

Telegram: @uni_k

www.konkur.in

when the page was added to DirtyPageTable and would be greater than or equal to the PageLSN for that page on stable storage.

19.9.2 Recovery Algorithm

ARIES recovers from a system crash in three passes.

- **Analysis pass**: This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.
- **Redo pass**: This pass starts from a position determined during analysis and performs a redo, repeating history, to bring the database to a state it was in before the crash.
- **Undo pass**: This pass rolls back all transactions that were incomplete at the time of crash.

19.9.2.1 Analysis Pass

The analysis pass finds the last complete checkpoint log record and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable. If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list.

The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass.

The analysis pass also updates DirtyPageTable whenever it finds a log record for an update on a page. If the page is not in DirtyPageTable, the analysis pass adds it to DirtyPageTable and sets the RecLSN of the page to the LSN of the log record.

19.9.2.2 Redo Pass

The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

Telegram: @uni_k

www.konkur.in

- If the page is not in DirtyPageTable or if the LSN of the update log record is less than the RecLSN of the page in DirtyPageTable, then the redo pass skips the log record.
- Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record.

Note that if either of the tests is negative, then the effects of the log record have already appeared on the page; otherwise the effects of the log record are not reflected on the page. Since ARIES allows non-idempotent physiological log records, a log record should not be redone if its effect is already reflected on the page. If the first test is negative, it is not even necessary to fetch the page from disk to check its PageLSN.

19.9.2.3 Undo Pass and Transaction Rollback

The undo pass is relatively straightforward. It performs a single backward scan of the log, undoing all transactions in undo-list. The undo pass examines only log records of transactions in undo-list; the last LSN recorded during the analysis pass is used to find the last log record for each transaction in undo-list.

Whenever an update log record is found, it is used to perform an undo (whether for transaction rollback during normal processing, or during the restart undo pass). The undo pass generates a CLR containing the undo action performed (which must be physiological). It sets the UndoNextLSN of the CLR to the PrevLSN value of the update log record.

If a CLR is found, its UndoNextLSN value indicates the LSN of the next log record to be undone for that transaction; later log records for that transaction have already been rolled back. For log records other than CLRs, the PrevLSN field of the log record indicates the LSN of the next log record to be undone for that transaction. The next log record to be processed at each stop in the undo pass is the maximum, across all transactions in undo-list, of next log record LSN.

Figure 19.10 illustrates the recovery actions performed by ARIES on an example log. We assume that the last completed checkpoint pointer on disk points to the checkpoint log record with LSN 7568. The PrevLSN values in the log records are shown using arrows in the figure, while the UndoNextLSN value is shown using a dashed arrow for the one compensation log record, with LSN 7565, in the figure. The analysis pass would start from LSN 7568, and when it is complete, RedoLSN would be 7564. Thus, the redo pass must start at the log record with LSN 7564. Note that this LSN is less than the LSN of the checkpoint log record, since the ARIES checkpointing algorithm does not flush modified pages to stable storage. The DirtyPageTable at the end of analysis would include pages 4894, 7200 from the checkpoint log record, and 2390 which is updated by the log record with LSN 7570. At the end of the analysis pass, the list of transactions to be undone consists of only T_{145} in this example.

The redo pass for the preceding example starts from LSN 7564 and performs redo of log records whose pages appear in DirtyPageTable. The undo pass needs to undo

Telegram: @uni_k

www.konkur.in

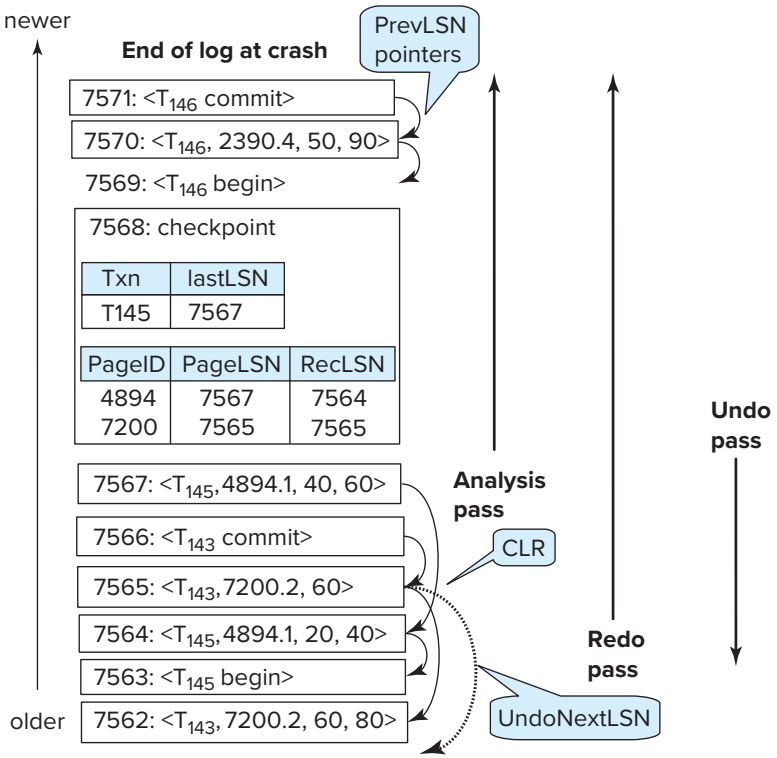


Figure 19.10 Recovery actions in ARIES.

only transaction T_{145}, and hence it starts from its LastLSN value 7567 and continues backwards until the record $<T_{145}$ start> is found at LSN 7563.

19.9.3 Other Features

Among other key features that ARIES provides are:

- **Nested top actions**: ARIES allows the logging of operations that should not be undone even if a transaction gets rolled back; for example, if a transaction allocates a page to a relation, even if the transaction is rolled back, the page allocation should not be undone since other transactions may have stored records in the page. Such operations that should not be undone are called nested top actions. Such operations can be modeled as operations whose undo action does nothing. In ARIES, such operations are implemented by creating a dummy CLR whose UndoNextLSN is set such that transaction rollback skips the log records generated by the operation.
- **Recovery independence**: Some pages can be recovered independently from others so that they can be used even while other pages are being recovered. If some pages

Telegram: @uni_k

www.konkur.in

of a disk fail, they can be recovered without stopping transaction processing on other pages.

- **Savepoints**: Transactions can record savepoints and can be rolled back partially up to a savepoint. This can be quite useful for deadlock handling, since transactions can be rolled back up to a point that permits release of required locks and then restarted from that point.

  Programmers can also use savepoints to undo a transaction partially, and then continue execution; this approach can be useful to handle certain kinds of errors detected during the transaction execution.

- **Fine-grained locking**: The ARIES recovery algorithm can be used with index concurrency-control algorithms that permit tuple-level locking on indices, instead of page-level locking, which improves concurrency significantly.

- **Recovery optimizations**: The DirtyPageTable can be used to prefetch pages during redo, instead of fetching a page only when the system finds a log record to be applied to the page. Out-of-order redo is also possible: Redo can be postponed on a page being fetched from disk and performed when the page is fetched. Meanwhile, other log records can continue to be processed.

In summary, the ARIES algorithm is a state-of-the-art recovery algorithm, incorporating a variety of optimizations designed to improve concurrency, reduce logging overhead, and reduce recovery time.

19.10 Recovery in Main-Memory Databases

Main-memory databases support fast querying and updates, since main memory supports very fast random access. However, the contents of main memory are lost on system failure, as well as on system shutdown. Thus, data must be additionally stored on persistent or stable storage to allow recovery of data when the system comes back up.

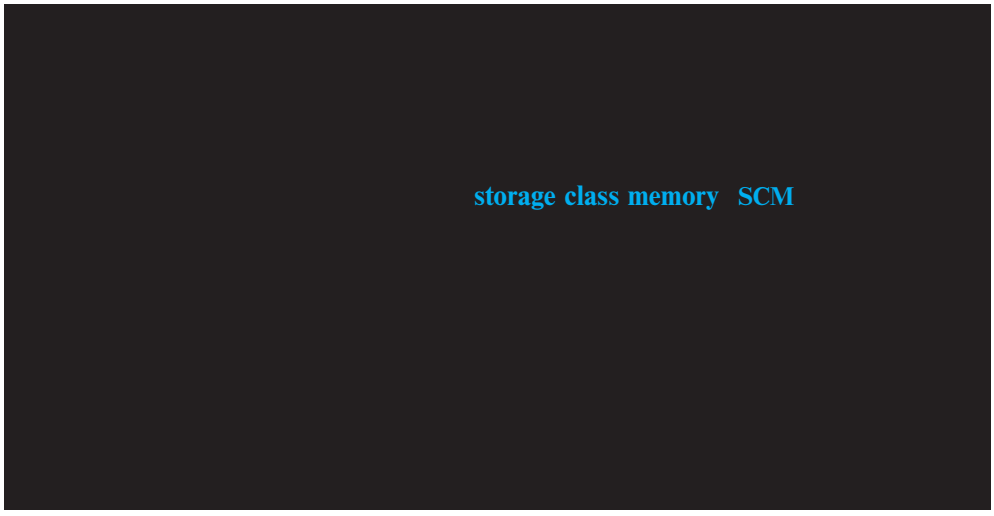
Traditional recovery algorithms can be used with main-memory databases. Log records for updates have to be output to stable storage. On recovery, the database has to be reloaded from disk and log records applied to restore the database state. Data blocks that have been modified by committed transactions still have to be written to disk, and checkpoints have to be performed, so that the amount of log that has to be replayed at recovery time is reduced.

However, some optimizations are possible with main-memory databases.

- With main-memory databases, indices can be rebuilt very quickly after the underlying relation is brought into memory and recovery has been performed on the relation. Thus, many systems do not perform any redo logging actions for index updates. Undo logging to support transaction abort is still required, but such undo

Telegram: @uni_k

www.konkur.in



log records can be kept in memory, and they need not be written to the log on stable storage.

- Several main-memory databases reduce logging overhead by performing only redo logging. Checkpoints are taken periodically, either ensuring that uncommitted data are not written to disk or avoiding in-place updates of records by creating multiple versions of records. Recovery consists of reloading the checkpoint and then performing redo operations. (Record versions created by uncommitted transactions must be garbage collected eventually.)
- Fast recovery is crucial for main-memory databases, since the entire database has to be loaded and recovery actions performed before any transaction processing can be done.

  Several main-memory databases therefore perform recovery in parallel using multiple cores, to minimize recovery time. To do so, data and log records may be partitioned, with log records of a partition affecting only data in the corresponding data partition. Each core is then responsible for performing recovery operations for a particular partition, and it can perform recovery operations in parallel with other cores.

19.11 Summary

- A computer system, like any other mechanical or electrical device, is subject to failure. There are a variety of causes of such failure, including disk crash, power failure, and software errors. In each of these cases, information concerning the database system is lost.

Telegram: @uni_k

www.konkur.in

- In addition to system failures, transactions may also fail for various reasons, such as violation of integrity constraints or deadlocks.
- An integral part of a database system is a recovery scheme that is responsible for the detection of failures and for the restoration of the database to a state that existed before the occurrence of the failure.
- The various types of storage in a computer are volatile storage, non-volatile storage, and stable storage. Data in volatile storage, such as in RAM, are lost when the computer crashes. Data in non-volatile storage, such as disk, are not lost when the computer crashes but may occasionally be lost because of failures such as disk crashes. Data in stable storage are never lost.
- Stable storage that must be accessible online is approximated with mirrored disks, or other forms of RAID, which provide redundant data storage. Offline, or archival, stable storage may consist of multiple tape copies of data stored in a physically secure location.
- In case of failure, the state of the database system may no longer be consistent; that is, it may not reflect a state of the world that the database is supposed to capture. To preserve consistency, we require that each transaction be atomic. It is the responsibility of the recovery scheme to ensure the atomicity and durability property.
- In log-based schemes, all updates are recorded on a log, which must be kept in stable storage. A transaction is considered to have committed when its last log record, which is the commit log record for the transaction, has been output to stable storage.
- Log records contain old values and new values for all updated data items. The new values are used in case the updates need to be redone after a system crash. The old values are used to roll back the updates of the transaction if the transaction aborts during normal operation, as well as to roll back the updates of the transaction in case the system crashed before the transaction committed.
- In the deferred-modifications scheme, during the execution of a transaction, all the write operations are deferred until the transaction has been committed, at which time the system uses the information on the log associated with the transaction in executing the deferred writes. With deferred modification, log records do not need to contain old values of updated data items.
- To reduce the overhead of searching the log and redoing transactions, we can use checkpointing techniques.
- Modern recovery algorithms are based on the concept of repeating history, whereby all actions taken during normal operation (since the last completed checkpoint) are replayed during the redo pass of recovery. Repeating history restores

Telegram: @uni_k

www.konkur.in

the system state to what it was at the time the last log record was output to stable storage before the system crashed. Undo is then performed from this state by executing an undo pass that processes log records of incomplete transactions in reverse order.

- Undo of an incomplete transaction writes out special redo-only log records and an abort log record. After that, the transaction can be considered to have completed, and it will not be undone again.
- Transaction processing is based on a storage model in which main memory holds a log buffer, a database buffer, and a system buffer. The system buffer holds pages of system object code and local work areas of transactions.
- Efficient implementation of a recovery scheme requires that the number of writes to the database and to stable storage be minimized. Log records may be kept in volatile log buffer initially, but they must be written to stable storage when one of the following conditions occurs:
    - Before the $<T_i$ commit> log record may be output to stable storage, all log records pertaining to transaction T_i must have been output to stable storage.
    - Before a block of data in main memory is output to the database (in non-volatile storage), all log records pertaining to data in that block must have been output to stable storage.
- Remote backup systems provide a high degree of availability, allowing transaction processing to continue even if the primary site is destroyed by a fire, flood, or earthquake. Data and log records from a primary site are continually backed up to a remote backup site. If the primary site fails, the remote backup site takes over transaction processing, after executing certain recovery actions.
- Modern recovery techniques support high-concurrency locking techniques, such as those used for B^+-tree concurrency control. These techniques allow early release of lower-level locks obtained by operations such as inserts or deletes, which allows other such operations to be performed by other transactions. After lower-level locks are released, physical undo is not possible, and instead logical undo, such as a deletion to undo an insertion, is required. Transactions retain higher-level locks that ensure that concurrent transactions cannot perform actions that could make logical undo of an operation impossible.
- To recover from failures that result in the loss of non-volatile storage, we must dump the entire contents of the database onto stable storage periodically—say, once per day. If a failure occurs that results in the loss of physical database blocks, we use the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, we use the log to bring the database system to the most recent consistent state.

Telegram: @uni_k

www.konkur.in

- The ARIES recovery scheme is a state-of-the-art scheme that supports a number of features to provide greater concurrency, reduce logging overheads, and minimize recovery time. It is also based on repeating history, and it allows logical undo operations. The scheme flushes pages on a continuous basis and does not need to flush all pages at the time of a checkpoint. It uses log sequence numbers (LSNs) to implement a variety of optimizations that reduce the time taken for recovery.

Review Terms

- Recovery scheme
- Failure classification
  - Transaction failure
  - Logical error
  - System error
  - System crash
  - Data-transfer failure
- Fail-stop assumption
- Disk failure
- Storage types
  - Volatile storage
  - Non-Volatile storage
  - Stable storage
- Blocks
  - Physical blocks
  - Buffer blocks
- Disk buffer
- Force-output
- Log-based recovery
- Log
- Log records
- Update log record
- Deferred modification
- Immediate modification
- Uncommitted modifications
- Checkpoints
- Recovery algorithm
- Restart recovery
- Transaction rollback
- Physical undo
- Physical logging
- Transaction rollback
- Restart recovery
- Redo phase
- Undo phase
- Repeating history
- Buffer management
- Log-record buffering
- Write-ahead logging (WAL)
- Log force
- Database buffering
- Latches
- Operating system and buffer management
- Fuzzy checkpointing
- High availability
- Remote backup systems
  - Primary site
  - Remote backup site
  - Secondary site

Telegram: @uni_k

www.konkur.in

- Detection of failure
- Transfer of control
- Time to recover
- Hot-spare configuration
- Time to commit
  - One-safe
  - Two-very-safe
  - Two-safe
- Early lock release
- Logical operations
- Logical logging
- Logical undo
- Loss of non-volatile storage
- Archival dump
- Fuzzy dump
- ARIES
  - Log sequence number (LSN)
  - PageLSN
  - Physiological redo
  - Compensation log record (CLR)
  - DirtyPageTable
  - Checkpoint log record
  - Analysis pass
  - Redo pass
  - Undo pass

Practice Exercises

19.1 Explain why log records for transactions on the undo-list must be processed in reverse order, whereas redo is performed in a forward direction.

19.2 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a media (disk) failure?

19.3 Some database systems allow the administrator to choose between two forms of logging: *normal logging*, used to recover from system crashes, and *archival logging*, used to recover from media (disk) failure. When can a log record be deleted, in each of these cases, using the recovery algorithm of Section 19.4?

19.4 Describe how to modify the recovery algorithm of Section 19.4 to implement savepoints and to perform rollback to a savepoint. (Savepoints are described in Section 19.9.3.)

19.5 Suppose the deferred modification technique is used in a database.

a. Is the old value part of an update log record required any more? Why or why not?

Telegram: @uni_k

www.konkur.in

b. If old values are not stored in update log records, transaction undo is clearly not feasible. How would the redo phase of recovery have to be modified as a result?

c. Deferred modification can be implemented by keeping updated data items in local memory of transactions and reading data items that have not been updated directly from the database buffer. Suggest how to efficiently implement a data item read, ensuring that a transaction sees its own updates.

d. What problem would arise with the above technique if transactions perform a large number of updates?

19.6 The shadow-paging scheme requires the page table to be copied. Suppose the page table is represented as a B^+-tree.

a. Suggest how to share as many nodes as possible between the new copy and the shadow copy of the B^+-tree, assuming that updates are made only to leaf entries, with no insertions or deletions.

b. Even with the above optimization, logging is much cheaper than a shadow copy scheme, for transactions that perform small updates. Explain why.

19.7 Suppose we (incorrectly) modify the recovery algorithm of Section 19.4 to note log actions taken during transaction rollback. When recovering from a system crash, transactions that were rolled back earlier would then be included in undo-list and rolled back again. Give an example to show how actions taken during the undo phase of recovery could result in an incorrect database state. (Hint: Consider a data item updated by an aborted transaction and then updated by a transaction that commits.)

19.8 Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.

19.9 Suppose a transaction deletes a record, and the free space generated thus is allocated to a record inserted by another transaction, even before the first transaction commits.

a. What problem can occur if the first transaction needs to be rolled back?

b. Would this problem be an issue if page-level locking is used instead of tuple-level locking?

c. Suggest how to solve this problem while supporting tuple-level locking, by logging post-commit actions in special log records, and executing

Telegram: @uni_k

www.konkur.in

them after commit. Make sure your scheme ensures that such actions are performed exactly once.

19.10 Explain the reasons why recovery of interactive transactions is more difficult to deal with than is recovery of batch transactions. Is there a simple way to deal with this difficulty? (Hint: Consider an automatic teller machine transaction in which cash is withdrawn.)

19.11 Sometimes a transaction has to be undone after it has committed because it was erroneously executed—for example, because of erroneous input by a bank teller.

a. Give an example to show that using the normal transaction undo mechanism to undo such a transaction could lead to an inconsistent state.

b. One way to handle this situation is to bring the whole database to a state prior to the commit of the erroneous transaction (called *point-in-time* recovery). Transactions that committed later have their effects rolled back with this scheme.

Suggest a modification to the recovery algorithm of Section 19.4 to implement point-in-time recovery using database dumps.

c. Later nonerroneous transactions can be reexecuted logically, if the updates are available in the form of SQL but cannot be reexecuted using their log records. Why?

19.12 The recovery techniques that we described assume that blocks are written atomically to disk. However, a block may be partially written when power fails, with some sectors written, and others not yet written.

a. What problems can partial block writes cause?

b. Partial block writes can be detected using techniques similar to those used to validate sector reads. Explain how.

c. Explain how RAID 1 can be used to recover from a partially written block, restoring the block to either its old value or to its new value.

19.13 The Oracle database system uses undo log records to provide a snapshot view of the database under snapshot isolation. The snapshot view seen by transaction T_i reflects updates of all transactions that had committed when T_i started and the updates of T_i; updates of all other transactions are not visible to T_i.

Describe a scheme for buffer handling whereby transactions are given a snapshot view of pages in the buffer. Include details of how to use the log to generate the snapshot view. You can assume that operations as well as their undo actions affect only one page.

Telegram: @uni_k

www.konkur.in

Exercises

19.14 Explain the difference between the three storage types—volatile, nonvolatile, and stable—in terms of I/O cost.

19.15 Stable storage cannot be implemented.

a. Explain why it cannot be.

b. Explain how database systems deal with this problem.

19.16 Explain how the database may become inconsistent if some log records pertaining to a block are not output to stable storage before the block is output to disk.

19.17 Outline the drawbacks of the no-steal and force buffer management policies.

19.18 Suppose two-phase locking is used, but exclusive locks are released early, that is, locking is not done in a strict two-phase manner. Give an example to show why transaction rollback can result in a wrong final state, when using the log-based recovery algorithm.

19.19 Physiological redo logging can reduce logging overheads significantly, especially with a slotted page record organization. Explain why.

19.20 Explain why logical undo logging is used widely, whereas logical redo logging (other than physiological redo logging) is rarely used.

19.21 Consider the log in Figure 19.5. Suppose there is a crash just before the log record $<T_0$ abort> is written out. Explain what would happen during recovery.

19.22 Suppose there is a transaction that has been running for a very long time but has performed very few updates.

a. What effect would the transaction have on recovery time with the recovery algorithm of Section 19.4, and with the ARIES recovery algorithm?

b. What effect would the transaction have on deletion of old log records?

19.23 Consider the log in Figure 19.7. Suppose there is a crash during recovery, just before the operation abort log record is written for operation O_1. Explain what will happen when the system recovers again.

19.24 Compare log-based recovery with the shadow-copy scheme in terms of their overheads for the case when data are being added to newly allocated disk pages (in other words, there is no old value to be restored in case the transaction aborts).

19.25 In the ARIES recovery algorithm:

Telegram: @uni_k

www.konkur.in

a. If at the beginning of the analysis pass, a page is not in the checkpoint dirty page table, will we need to apply any redo records to it? Why?

b. What is RecLSN, and how is it used to minimize unnecessary redos?

19.26 Explain the difference between a system crash and a "disaster."

19.27 For each of the following requirements, identify the best choice of degree of durability in a remote backup system:

a. Data loss must be avoided, but some loss of availability may be tolerated.

b. Transaction commit must be accomplished quickly, even at the cost of loss of some committed transactions in a disaster.

c. A high degree of availability and durability is required, but a longer running time for the transaction commit protocol is acceptable.

Further Reading

[Gray and Reuter (1993)] is an excellent textbook source of information about recovery, including interesting implementation and historical details. [Bernstein and Goodman (1981)] is an early textbook source of information on concurrency control and recovery. [Faerber et al. (2017)] provide an overview of main-memory databases, including recovery techniques.

An overview of the recovery scheme of System R is presented by [Gray (1978)] (which also includes extensive coverage of concurrency control and other aspects of System R), and [Gray et al. (1981)]. A comprehensive presentation of the principles of recovery is offered by [Haerder and Reuter (1983)]. The ARIES recovery method is described in [Mohan et al. (1992)]. Many databases support high-availability features; more details may be found in their online manuals.

Bibliography

[Bayer et al. (1978)] R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, Springer Verlag (1978).

[Bernstein and Goodman (1981)] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Volume 13, Number 2 (1981), pages 185–221.

[Faerber et al. (2017)] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo, "Main Memory Database Systems", *Foundations and Trends in Databases*, Volume 8, Number 1-2 (2017), pages 1–130.

[Gray (1978)] J. Gray. "Notes on Data Base Operating System", In *[Bayer et al. (1978)]*, pages 393–481. Springer Verlag (1978).

Telegram: @uni_k

www.konkur.in

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Gray et al. (1981)] J. Gray, P. R. McJones, and M. Blasgen, "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys*, Volume 13, Number 2 (1981), pages 223–242.

[Haerder and Reuter (1983)] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys*, Volume 15, Number 4 (1983), pages 287–318.

[Mohan et al. (1992)] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, Volume 17, Number 1 (1992), pages 94–162.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



PART 8

PARALLEL AND DISTRIBUTED DATABASES

Database systems can be centralized, where one server machine executes operations on the database. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

Chapter 20 first outlines the architectures of database systems running on server systems, which are used in centralized and client–server architectures. The chapter then outlines parallel computer architectures, and parallel database architectures designed for different types of parallel computers. Finally, the chapter outlines architectural issues in building a distributed database system.

Chapter 21 discusses techniques for data storage and indexing in parallel and distributed database systems. These include data partitioning and replication. Key-value stores, which offer some but not all features of a full database system, are discussed along with their benefits and drawbacks.

Chapter 22 discusses algorithms for query processing in parallel and distributed database systems. This chapter focuses on query processing in decision-support systems. Such systems need to execute queries on very large amounts of data, and parallel processing of the query across multiple nodes is critical for processing queries within acceptable response times. The chapter covers parallel sort and join, pipelining, the implementation of MapReduce systems, and parallel stream processing.

Chapter 23 discusses how to carry out transaction processing in parallel and distributed database systems. In addition to supporting concurrency control and recovery, the system must deal with issues pertaining to replication of data and with failures that involve some, but not all, nodes. The chapter covers atomic commit protocols and consensus protocols designed for distributed databases, distributed concurrency control, replica consistency, and trade-offs of consistency for the sake of performance and availability.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 20



Database-System Architectures

The architecture of a database system is greatly influenced by the underlying computer system on which it runs, in particular by such aspects as processor and memory architecture, and networking, as well as by requirements of parallelism and distribution. In this chapter, we provide a high-level view of database architectures, with a focus on how they are influenced by the underlying hardware, as well as by requirements of parallel and distributed processing.

20.1 Overview

The earliest databases were built to run on a single physical machine supporting multitasking; such *centralized database systems* are still widely used. An enterprise-scale application that runs on a centralized database system today may have from tens to thousands of users and database sizes ranging from megabytes to hundreds of gigabytes.

Parallel database systems were developed, starting in the late 1980s to execute tasks in parallel on a large number of machines. These were developed to handle high-end enterprise applications whose requirements in terms of transaction processing performance, time to process decision support queries, and storage capacity could not be met by centralized databases. These databases were designed to run in parallel on hundreds of machines. Today, the growth of parallel databases is driven not just by enterprise applications, but even more so by web-scale applications, which may have millions to even hundreds of millions of users and may need to deal with many petabytes of data.

Parallel data storage systems are designed primarily to store and retrieve data based on keys. Unlike parallel databases, data storage systems typically provide very limited support for transactions, and they lack support for declarative querying. On the other hand, such systems can be run in parallel on very large numbers of machines (thousands to tens of thousands), a scale that most parallel databases cannot handle.

Further, data are often generated and stored on different database systems, and there is a need to execute queries and update transactions across multiple databases. This need led to the development of *distributed database systems*. Techniques developed for fault tolerance in the context of distributed databases today also play a key role in

Telegram: @uni_k

www.konkur.in

ensuring the extremely high reliability and availability of massively parallel database and data storage systems.

We study the architecture of database systems in this chapter, starting with the traditional centralized architectures and covering parallel and distributed database architectures. We cover issues of parallel and distributed data storage and indexing in Chapter 21. Chapter 22 covers issues of parallel and distributed query processing, while Chapter 23 covers issues of parallel and distributed transaction processing.

20.2 Centralized Database Systems

Centralized database systems are those that run on a single computer system. Such database systems span a range from single-user database systems running on mobile devices or personal computers to high-performance database systems running on a server with multiple CPU cores and disks and a large amount of main memory that can be accessed by any of the CPU cores. Centralized database systems are widely used for enterprise-scale applications.

We distinguish two ways in which computers are used: as single-user systems and as multiuser systems. Smartphones and personal computers fall into the first category. A typical **single-user system** is a system used by a single person, usually with only one processor (usually with multiple cores), and one or two disks.[1] A typical **multiuser system**, on the other hand, has multiple disks, a large amount of memory, and multiple processors. Such systems serve a large number of users who are connected to the system remotely, and they are called **server systems**.

Database systems designed for single-user systems usually do not provide many of the facilities that a multiuser database provides. In particular, they may support very simple concurrency control schemes, since highly concurrent access to the database is very unlikely. Provisions for crash recovery in such systems may also be either very basic (e.g., making a copy of data before updating it), or even absent in some cases. Such systems may not support SQL and may instead provide an API for data access. Such database systems are referred to as **embedded databases**, since they are usually designed to be linked to a single application program and are accessible only from that application.

In contrast, multiuser database systems support the full transactional features that we have studied earlier. Such databases are usually designed as **servers**, which service requests received from application programs; the requests could be in the form of SQL queries, or they could be requests for retrieving, storing, or updating data specified using an API.

Most general-purpose computer systems in use today have a few multicore processors (typically one to four), with each multicore processor having a few cores (typically

[1]Recall that we use the term *disk* to refer to hard disks as well as solid-state

Telegram: @uni_k

www.konkur.in

4 to 8). Main memory is shared across all the processors. Parallelism with such a small number of cores, and with shared memory, is referred to as **coarse-grained parallelism**.

Operating systems that run on single-processor systems support multitasking, allowing multiple processes to run on the same processor in a time-shared manner. Actions of different processes may thus be interleaved. Databases designed for single-processor machines have long been designed to allow multiple processes or threads to access shared database structures concurrently. Thus, many of the issues in handling multiple processes running truly in parallel, such as concurrent access to data, are already addressed by databases designed for single-processor machines. As a result, database systems designed for time-shared single-processor machines could be adapted relatively easily to run on coarse-grained parallel systems.

Databases running on coarse-grained parallel machines traditionally did not attempt to partition a single query among the processors; instead, they ran each query on a single processor, allowing multiple queries to run concurrently. Thus, such systems support a higher throughput; that is, they allow a greater number of transactions to run per second, although individual transactions do not run any faster. In recent years, with even mobile phones supporting multiple cores, such systems are evolving to support parallel processing of individual queries.

In contrast, machines with **fine-grained parallelism** have a large number of processors, and database systems running on such machines attempt to parallelize single tasks (queries, for example) submitted by users.

Parallelism has emerged as a critical issue in the design of software systems in general, and in particular in the design of database systems. As a result, parallel database systems, which once were specialized systems running on specially designed hardware, are increasingly becoming the norm. We study the architecture of parallel database systems in Section 20.4.

20.3 Server System Architectures

Server systems can be broadly categorized as transaction servers and data servers.

- **Transaction-server** systems, also called **query-server** systems, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client. Usually, client machines ship transactions to the server systems, where those transactions are executed, and results are shipped back to clients that are in charge of displaying the data. Requests may be specified through the use of SQL or through a specialized application program interface.
- **Data-server systems** allow clients to interact with the servers by making requests to read or update data, in units such as files, pages, or objects. For example, file servers provide a file-system interface where clients can create, update, read, and

Telegram: @uni_k

www.konkur.in

delete files. Data servers for database systems offer much more functionality; they support units of data—such as pages, tuples, or objects—that are smaller than a file. They provide indexing facilities for data, and they provide transaction facilities so that the data are never left in an inconsistent state if a client machine or process fails.

Of these, the transaction-server architecture is by far the more widely used architecture, although parallel data servers are widely used to handle traffic at web scale. We shall elaborate on the transaction-server and data-server architectures in Section 20.3.1 and Section 20.3.2.

20.3.1 Transaction-Server Architecture

A typical transaction-server system today consists of multiple processes accessing data in shared memory, as in Figure 20.1. The processes that form part of the database system include:

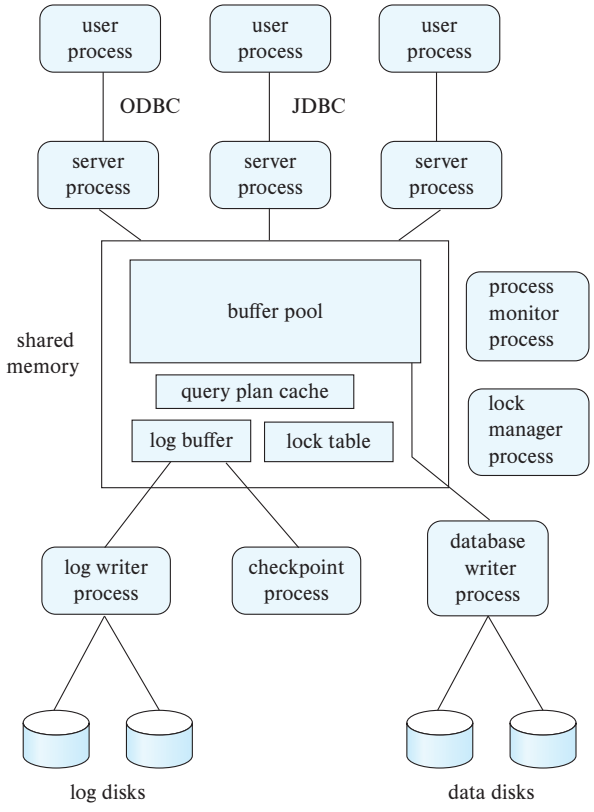


Figure 20.1 Shared memory and process structure.

Telegram: @uni_k

www.konkur.in

- **Server processes**: These are processes that receive user queries (transactions), execute them, and send the results back. The queries may be submitted to the server processes from a user interface, or from a user process running embedded SQL, or via JDBC, ODBC, or similar protocols. Some database systems use a separate process for each user session, and a few use a single database process for all user sessions, but with multiple threads so that multiple queries can execute concurrently. (A **thread** is similar to a process, but multiple threads execute as part of the same process, and all threads within a process run in the same virtual-memory space. Multiple threads within a process can execute concurrently.) Many database systems use a hybrid architecture, with multiple processes, each one running multiple threads.
- **Lock manager process**: This process implements lock manager functionality, which includes lock grant, lock release, and deadlock detection.
- **Database writer process**: There are one or more processes that output modified buffer blocks back to disk on a continuous basis.
- **Log writer process**: This process outputs log records from the log record buffer to stable storage. Server processes simply add log records to the log record buffer in shared memory, and if a log force is required, they request the log writer process to output log records (recall that a log force causes the log contents in memory to be output to stable storage).
- **Checkpoint process**: This process performs periodic checkpoints.
- **Process monitor process**: This process monitors other processes, and if any of them fails, it takes recovery actions for the process, such as aborting any transaction being executed by the failed process and then restarting the process.

The shared memory contains all shared data, such as:

- Buffer pool.
- Lock table.
- Log buffer, containing log records waiting to be output to the log on stable storage.
- Cached query plans, which can be reused if the same query is submitted again.

All database processes can access the data in shared memory. Since multiple processes may read or perform updates on data structures in shared memory, there must be a mechanism to ensure **mutual exclusion**, that is, to ensure that a data structure is modified by at most one process at a time, and no process is reading a data structure while it is being written by other processes.

Such mutual exclusion can be implemented by means of operating system functions called semaphores. Alternative implementations, with less overhead, use one of

Telegram: @uni_k

www.konkur.in

1. test-and-set

2. compare-and-swap

Telegram: @uni_k

www.konkur.in

the **atomic instructions**, test-and-set, or compare-and-swap, which are supported by the computer hardware. See Note 20.1 on page 966 for details of these instructions. All multiprocessor systems today support either the test-and-set or the compare-and-swap atomic instructions. Further details on these instructions may be found in operating systems textbooks.

Note that the atomic instructions can be used for mutual exclusion, which is equivalent to supporting exclusive locks, but they do not directly support shared locks. Thus, they cannot be used directly to implement general-purpose locking in databases. Atomic instructions are, however, used to implement short-duration locks, also known as latches, which are used for mutual exclusion in databases.

To avoid the overhead of message passing, in many database systems, server processes implement locking by directly updating the lock table (which is in shared memory) instead of sending lock request messages to a lock manager process. (The lock table is shown in Figure 18.10.)

Since multiple server processes may access the lock table in shared memory concurrently, processes must ensure mutual exclusion on access to the lock table. This is typically done by acquiring a mutex (also referred to as a latch) on the lock table, using the test-and-set or compare-and-swap instructions on a memory location representing a lock on the lock table.

A transaction that wants to acquire a lock by directly updating the lock table in shared memory executes the following steps.

1. Acquire a mutex (latch) on the lock table.
2. Check if the requested lock can be allocated, using the procedure we saw in Section 18.1.4. If it can, update the lock table to indicate the lock is allocated. Otherwise, update the lock table to indicate that the lock request is in the queue for that lock.
3. Release the mutex on the lock table.

If a lock cannot be obtained immediately because of a lock conflict, the transaction may periodically read the lock table to check if the lock has been allocated to it due to a lock release, which is described next.

Lock release is done as follows:

1. Acquire a mutex on the lock table
2. Remove the entry in the lock table for the lock being released.
3. If there are any other lock requests pending for the data item that can now be allocated to the lock, the lock table is updated to mark those requests as allocated. The rules on which lock requests may be granted are as described in Section 18.1.4.
4. Release the mutex on the lock table.

Telegram: @uni_k

www.konkur.in

To avoid repeated checks on the lock table (an example of the phenomenon of *busy waiting*), operating system semaphores can be used by the lock request code to wait for a lock grant notification. The lock release code must then use the semaphore mechanism to notify waiting transactions that their locks have been granted.

Even if the system handles lock requests through shared memory, it still uses the lock manager process for deadlock detection.

20.3.2 Data Servers and Data Storage Systems

Data-server systems were originally developed to support data access from object-oriented databases; object-oriented databases allow programmers to use a programming language that allows creation, retrieval, and update of persistent objects.

Many of the target applications of object-oriented databases, such as computer-aided design (CAD) systems, required extensive computation on the retrieved data. For example, the CAD system may store a model of a computer chip or a building, and it may perform computations such as simulations on the retrieved model, which may be expensive in terms of CPU time.

If all the computation were done at the server, the server would be overloaded. Instead, in such an environment, it makes sense to store data on a separate data server machine, fetch data to client machines when needed, perform all processing at the client machines, and then to store new or updated data back on the data server machine. Thus, the processing power of client machines can be used to carry out the computation, while the server needs only to store and fetch data, without performing any computation.

More recently, a number of parallel data storage systems have been developed for handling very high volumes of data and transactions. Such systems do not necessarily support SQL, but instead provide APIs for storing, retrieving, and updating data items. Data items stored in such systems they could be tuples, or could be objects represented in formats such as JSON or XML, or they could even be files or documents.

We use the term **data item** to refer to tuples, objects, files, and documents. We also use the terms *data server* and *data storage system* interchangeably.

Data servers support communication of entire data items; in the case of very large data items, they may also support communication of only specified parts of the data item, for instance, specified blocks, instead of requiring that the entire data item be fetched or stored.

Data servers in earlier generations of storage systems supported a concept called *page shipping*, where the unit of communication is a database page that may potentially contain multiple data items. Page shipping is not used today, since storage systems do not expose the underlying storage layout to clients.

20.3.3 Caching at Clients

The time cost of communication between a client application and a server (whether a transaction server, or a data server) is high compared to that of a local memory

Telegram: @uni_k

www.konkur.in

reference (milliseconds, versus less than 100 nanoseconds). The time to send a message over a network, and get a response back, called the *network round-trip time*, or *network latency*, can be nearly a millisecond even if the data server is in the same location as the client.

As a result, applications running at the clients adopt several optimization strategies to reduce the effects of network latency. The same strategies can also be useful in parallel database systems, where some of the data required for processing a query may be stored on a different machine from where it is consumed. The optimization strategies include the following:

- **Prefetching**. If the unit of communication is a single small item, the overhead of message passing is high compared to the amount of data transmitted. In particular, network latency can cause significant delays if a transaction makes repeated requests for data items across a network.

  Thus, when an item is requested, it may make sense to also send other items that are likely to be used in the near future. Fetching items even before they are requested is called **prefetching**.

- **Data caching**. Data that are shipped to a client on behalf of a transaction can be **cached** at the client within the scope of a single transaction. Data can be cached even after the transaction completes, allowing successive transactions at the same client to make use of the cached data.

  However, **cache coherency** is an issue: Even if a transaction finds cached data, it must make sure that those data are up to date, since they may have been updated, or even deleted, by a different client after they were cached. Thus, a message must still be exchanged with the server to check validity of the data and to acquire a lock on the data, unless the application is willing to live with potentially stale data. Further, new tuples may have been inserted after a transaction caches data, which may not be in the cache. The transaction may have to contact the server to find such tuples.

- **Lock caching**. If the usage of data is mostly partitioned among the clients, with clients rarely requesting data that are also requested by other clients, locks can also be cached at the client machine. Suppose that a client finds a data item in the cache, and that it also finds the lock required for an access to the data item in the cache. Then, the access can proceed without any communication with the server. However, the server must keep track of cached locks; if a client requests a lock from the server, the server must **call back** all conflicting locks on the data item from any other client machines that have cached the locks. The task becomes more complicated when machine failures are taken into account.

- **Adaptive lock granularity**. If a transaction requires locks on multiple data items, discovered in the course of a transaction, and each lock acquisition requires a round trip to a data server, the transaction may waste a good deal of time on

Telegram: @uni_k

www.konkur.in

just lock acquisition. In such a situation, multi-granularity locking can be used to avoid multiple requests. For example, if multiple data items are stored in a page, a single page lock (which is at a coarser granularity) can avoid the need to acquire multiple data-item locks (which are at a finer granularity). This strategy works well if there is very little lock contention, but with higher contention, acquiring a coarse granularity lock can affect concurrency significantly.

Lock de-escalation, is a way of adaptively decreasing the lock granularity if there is higher contention. Lock de-escalation is initiated by the data server sending a request to the client to de-escalate a lock, and the client responds by acquiring finer-granularity locks and then releasing the coarser-granularity lock.

When switching to a finer granularity, if some of the locks were for cached data items that are not currently locked by any transaction at a client, the data item can be removed from the cache instead of acquiring a finer-granularity lock on it.

20.4 Parallel Systems

Parallel systems improve processing and I/O speeds by using a large number of computers in parallel. Parallel machines are becoming increasingly common, making the study of parallel database systems correspondingly more important.

In parallel processing, many operations are performed simultaneously, as opposed to serial processing, in which the computational steps are performed sequentially. A **coarse-grain** parallel machine consists of a small number of powerful processors; a **massively parallel** or **fine-grain parallel** machine uses thousands of smaller processors. Virtually all high-end server machines today offer some degree of coarse-grain parallelism, with up to two or four processors each of which may have 20 to 40 cores.

Massively parallel computers can be distinguished from the coarse-grain parallel machines by the much larger degree of parallelism that they support. It is not practical to share memory between a large number of processors. As a result, massively parallel computers are typically built using a large number of computers, each of which has its own memory, and often, its own set of disks. Each such computer is referred to as a **node** in the system.

Parallel systems at the scale of hundreds to thousands of nodes or more are housed in a **data center**, which is a facility that houses a large number of servers. Data centers provide high-speed network connectivity within the data center, as well as to the outside world. The numbers and sizes of data centers have grown tremendously in the last decade, and modern data centers may have several hundred thousand servers.

20.4.1 Motivation for Parallel Databases

The transaction requirements of organizations have grown with the increasing use of computers. Moreover, the growth of the World Wide Web has created many sites with millions of viewers, and the increasing amounts of data collected from these viewers has produced extremely large databases at many companies.

Telegram: @uni_k

www.konkur.in

The driving force behind parallel database systems is the demands of applications that have to query extremely large databases (of the order of petabytes—that is, 1000 terabytes, or equivalently, 10^{15} bytes) or that have to process an extremely large number of transactions per second (of the order of thousands of transactions per second). Centralized and client–server database systems are not powerful enough to handle such applications.

Web-scale applications today often require hundreds to thousands of nodes (and in some cases, tens of thousands of nodes) to handle the vast number of users on the web.

Organizations are using these increasingly large volumes of data—such as data about what items people buy, what web links users click on, and when people make telephone calls—to plan their activities and pricing. Queries used for such purposes are called **decision-support queries**, and the data requirements for such queries may run into terabytes. Single-node systems are not capable of handling such large volumes of data at the required rates.

The set-oriented nature of database queries naturally lends itself to parallelization. A number of commercial and research systems have demonstrated the power and scalability of parallel query processing.

As the cost of computing systems has reduced significantly over the years, parallel machines have become common and relatively inexpensive. Individual computers have themselves become parallel machines using multicore architectures. Parallel databases are thus quite affordable even for small organizations.

Parallel database systems which can support hundreds of nodes have been available commercially for several decades, but the number of such products has seen a significant increase since the mid 2000s. Open-source platforms for parallel data storage such as the Hadoop File System (HDFS), and HBase, and for query processing, such as Hadoop Map-Reduce and Hive (among many others), have also seen extensive adoption.

It is worth noting that application programs are typically built such that they can be executed in parallel on a number of application servers, which communicate over a network with a database server, which may itself be a parallel system. The parallel architectures described in this section can be used not only for data storage and query processing in the database but also for parallel processing of application programs.

20.4.2 Measures of Performance for Parallel Systems

There are two main measures of performance of a database system: (1) **throughput**, the number of tasks that can be completed in a given time interval, and (2) **response time**, the amount of time it takes to complete a single task from the time it is submitted. A system that processes a large number of small transactions can improve throughput by processing many transactions in parallel. A system that processes large transactions can improve response time as well as throughput by performing subtasks of each transaction in parallel.

Telegram: @uni_k

www.konkur.in

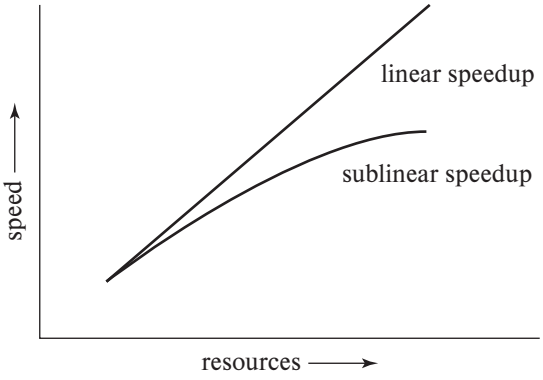


Figure 20.2 Speedup with increasing resources.

Parallel processing within a computer system allows database-system activities to be speeded up, allowing faster response to transactions, as well as more transactions to be executed per second. Queries can be processed in a way that exploits the parallelism offered by the underlying computer system.

Two important issues in studying parallelism are speedup and scaleup. Running a given task in less time by increasing the degree of parallelism is called **speedup**. Handling larger tasks by increasing the degree of parallelism is called **scaleup**.

Consider a database application running on a parallel system with a certain number of processors and disks. Now suppose that we increase the size of the system by increasing the number of processors, disks, and other components of the system. The goal is to process the task in time inversely proportional to the number of processors and disks allocated. Suppose that the execution time of a task on the larger machine is T_L, and that the execution time of the same task on the smaller machine is T_S. The speedup due to parallelism is defined as T_S/T_L. The parallel system is said to demonstrate **linear speedup** if the speedup is N when the larger system has N times the resources (processors, disk, and so on) of the smaller system. If the speedup is less than N, the system is said to demonstrate **sublinear speedup**. Figure 20.2 illustrates linear and sublinear speedup.[2]

Scaleup relates to the ability to process larger tasks in the same amount of time by providing more resources. Let Q be a task, and let Q_N be a task that is N times bigger than Q. Suppose that the execution time of task Q on a given machine M_S is T_S, and the execution time of task Q_N on a parallel machine M_L that is N times larger than M_S is T_L. The scaleup is then defined as T_S/T_L. The parallel system M_L is said to demonstrate **linear scaleup** on task Q if $T_L = T_S$. If $T_L > T_S$, the system is said

[2]In some cases, a parallel system may provide superlinear speedup, that is, an N times larger system may provide speedup greater than N. This could happen, for example, because data that did not fit in the main memory of a smaller system do fit in the main memory of a larger system, avoiding disk I/O. Similarly, data may fit in the cache of a larger system, reducing memory accesses compared to a smaller system, which

Telegram: @uni_k

www.konkur.in

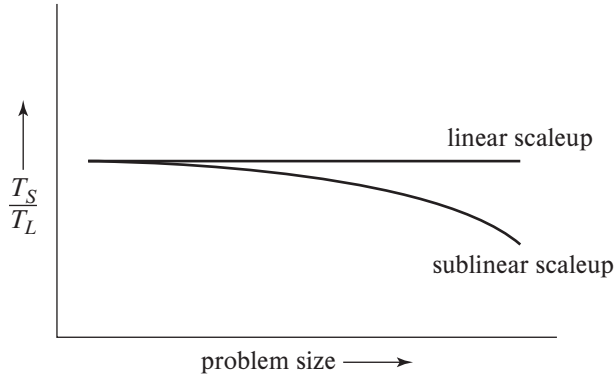


Figure 20.3 Scaleup with increasing problem size and resources.

to demonstrate **sublinear scaleup**. Figure 20.3 illustrates linear and sublinear scaleups (where the resources increase in proportion to problem size). There are two kinds of scaleup that are relevant in parallel database systems, depending on how the size of the task is measured:

- In **batch scaleup**, the size of the database increases, and the tasks are large jobs whose runtime depends on the size of the database. An example of such a task is a scan of a relation whose size is proportional to the size of the database. Thus, the size of the database is the measure of the size of the problem. Batch scaleup also applies in scientific applications, such as executing a weather simulation at an N-times finer resolution,[3] or performing the simulation for an N-times longer period of time.
- In **transaction scaleup**, the rate at which transactions are submitted to the database increases, and the size of the database increases proportionally to the transaction rate. This kind of scaleup is what is relevant in transaction-processing systems where the transactions are small updates—for example, a deposit or withdrawal from an account—and transaction rates grow as more accounts are created. Such transaction processing is especially well adapted for parallel execution, since transactions can run concurrently and independently on separate nodes, and each transaction takes roughly the same amount of time, even if the database grows.

Scaleup is usually the more important metric for measuring the efficiency of parallel database systems. The goal of parallelism in database systems is usually to make sure that the database system can continue to perform at an acceptable speed, even as the size of the database and the number of transactions increases. Increasing the ca-

[3]For example, a weather simulation that divides the atmosphere in a particular region into cubes of side 200 meters may need to be modified to use a finer resolution, with cubes of side 100 meters; the number of cubes would thus be scaled up by a factor of 8.

Telegram: @uni_k

www.konkur.in

pacity of the system by increasing the parallelism provides a smoother path for growth for an enterprise than does replacing a centralized system with a faster machine (even assuming that such a machine exists). However, we must also look at absolute performance numbers when using scaleup measures; a machine that scales up linearly may perform worse than a machine that scales less than linearly, simply because the latter machine is much faster to start off with.

A number of factors work against efficient parallel operation and can diminish both speedup and scaleup.

- **Sequential computation**. Many tasks have some components that can benefit from parallel processing, and some components that have to be executed sequentially. Consider a task that takes time T to run sequentially. Suppose the fraction of the total execution time that can benefit from parallelization is p, and that part is executed by n nodes in parallel. Then the total time taken would be $(1-p)T+(p/n)T$, and the speedup would be $\frac{1}{(1-p)+(p/n)}$. (This formula is referred to as **Amdahl's law**.) If the fraction p is, say $\frac{9}{10}$, then the maximum speedup possible, even with very large n, would be 10.

  Now consider scaleup, where the problem size increases. If the time taken by the sequential part of a task increases along with the problem size, scaleup will be similarly limited. Suppose fraction p of the execution time of a problem benefits from increasing resources, while fraction $(1-p)$ is sequential and does not benefit from increasing resources. Then the scaleup with n times more resources on a problem that is n times larger will be $\frac{1}{n(1-p)+p}$. (This formula is referred to as **Gustafson's law**.) However, if the time taken by the sequential part does not increase with problem size, its impact on scaleup will be less as the problem sizes.

  Start-up costs. There is a start-up cost associated with initiating a single process. In a parallel operation consisting of thousands of processes, the *start-up time* may overshadow the actual processing time, affecting speedup adversely.

- **Interference**. Since processes executing in a parallel system often access shared resources, a slowdown may result from the *interference* of each new process as it competes with existing processes for commonly held resources, such as a system bus, or shared disks, or even locks. Both speedup and scaleup are affected by this phenomenon.

- **Skew**. By breaking down a single task into a number of parallel steps, we reduce the size of the average step. Nonetheless, the service time for the single slowest step will determine the service time for the task as a whole. It is often difficult to divide a task into exactly equal-sized parts, and the way that the sizes are distributed is therefore *skewed*. For example, if a task of size 100 is divided into 10 parts, and the division is skewed, there may be some tasks of size less than 10 and some tasks of size more than 10; if even one task happens to be of size 20, the speedup obtained by running the tasks in parallel is only 5, instead of 10 as we would have hoped.

Telegram: @uni_k

www.konkur.in

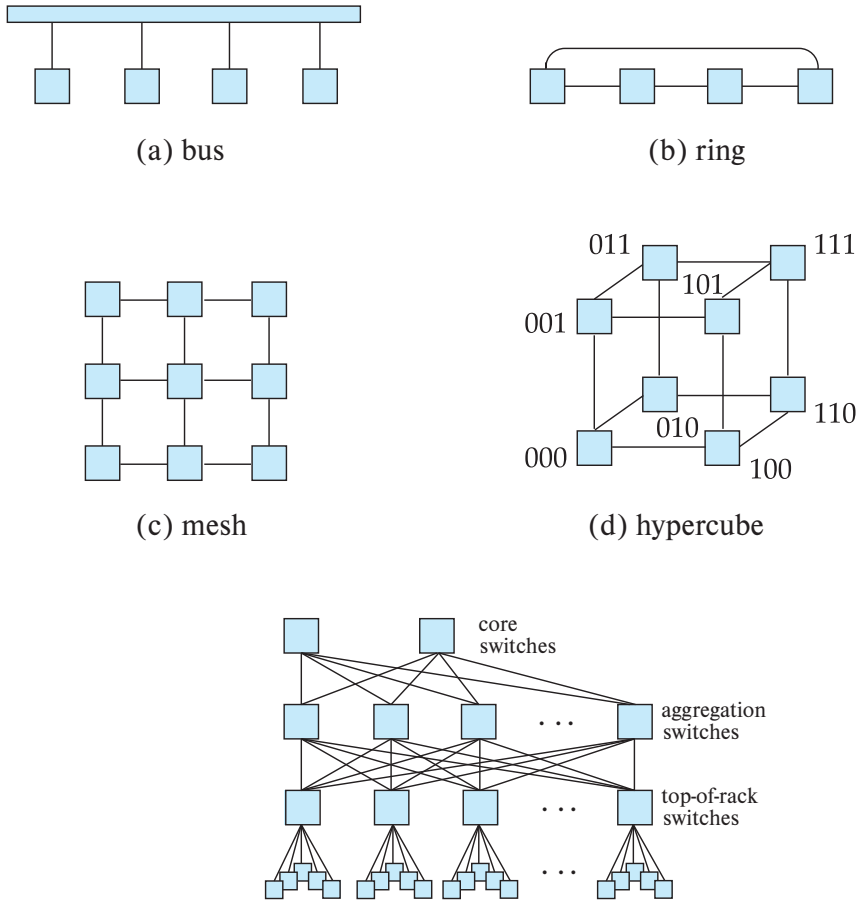


Figure 20.4 Interconnection networks.

20.4.3 Interconnection Networks

Parallel systems consist of a set of components (processors, memory, and disks) that can communicate with each other via an **interconnection network**. Figure 20.4 shows several commonly used types of interconnection networks:

- **Bus**. All the system components can send data on and receive data from a single communication bus. This type of interconnection is shown in Figure 20.4a. Bus interconnects were used in earlier days to connect multiple nodes in a network, but they are no longer used for this task. However, bus interconnections are still used for connecting multiple CPUs and memory units within a single node, and they work well for small numbers of processors. However, they do not scale well

Telegram: @uni_k

www.konkur.in

with increasing parallelism, since the bus can handle communication from only one component at a time; with increasing numbers of CPUs and memory banks in a node, other interconnection mechanisms such as ring or mesh interconnections are now used even within a single node.

- **Ring**. The components are nodes arranged in a ring (circle), and each node is connected to its two adjacent nodes in the ring, as shown in Figure 20.4b. Unlike a bus, each link can transmit data concurrently with other links in the ring, leading to better scalability. However, to transmit data from one node to another node on the ring may require a large number of hops; specifically, up to $n/2$ hops may be needed on a ring with n nodes, assuming communication can be done in either direction on the ring. Furthermore, the transmission delay increases if the number of nodes in the ring is increased.

- **Mesh**. The components are nodes in a grid, and each component connects to all its adjacent components in the grid. In a two-dimensional mesh, each node connects to (up to) four adjacent nodes, while in a three-dimensional mesh, each node connects to (up to) six adjacent nodes. Figure 20.4c shows a two-dimensional mesh. Nodes that are not directly connected can communicate with one another by routing messages via a sequence of intermediate nodes that are directly connected to one another. The number of communication links grows as the number of components grows, and the communication capacity of a mesh therefore scales better with increasing parallelism.

  Mesh interconnects are used to connect multiple cores in a processor, or processors in a single server, to each other; each processor core has direct access to a bank of memory connected to the processor core, but the system transparently fetches data from other memory banks by sending messages over the mesh interconnects.

  However, mesh interconnects are no longer used for interconnecting nodes, since the number of hops required to transmit data increases significantly with the number of nodes (the number of hops required to transmit data from one node to another node in a mesh is proportional in the worst case to the square root of the number of nodes). Parallel systems today have very large numbers of nodes, and mesh interconnects would thus be impractically slow.

- **Hypercube**. The components are numbered in binary, and a component is connected to another if the binary representations of their numbers differ in exactly one bit. Thus, each of the n components is connected to $\log(n)$ other components. Figure 20.4d shows a hypercube with eight nodes. In a hypercube interconnection, a message from a component can reach any other component by going through at most $\log(n)$ links. In contrast, in a mesh architecture a component may be $2(\sqrt{n} - 1)$ links away from some of the other components (or $\sqrt{n}$ links away, if the mesh interconnection wraps around at the edges of the grid). Thus communication delays in a hypercube are significantly lower than in a mesh.

Telegram: @uni_k

www.konkur.in

Hypercubes have been used to interconnect nodes in massively parallel computers in earlier days, but they are no longer commonly used.

- **Tree-like**. Server systems in a data center are typically mounted in racks, with each rack holding up to about 40 nodes. Multiple racks are used to build systems with larger numbers of nodes. A key issue is how to interconnect such nodes.

  To connect nodes within a rack, there is typically a network switch mounted at the top of the rack; 48 port switches are commonly used, so a single switch can be used to connect all the servers in a rack. Current-generation network switches typically support a bandwidth of 1 to 10 gigabits per second (Gbps) simultaneously from/to each of the servers connected to the switch, although more expensive network interconnects with 40 to 100 Gbps bandwidths are available.

  Multiple top-of-rack switches (also referred to as **edge switches**) can in turn be connected to another switch, called an **aggregation switch**, allowing interconnection between racks. If there are a large number of racks, the racks may be divided into groups, with one aggregation switch connecting a group of racks, and all the aggregation switches in turn connected to a **core switch**. Such an architecture is a **tree topology** with three tiers. The core switch at the top of the tree also provides connectivity to outside networks.

  A problem with this basic tree structure, which is frequently used in local-area networks within organizations, is that the available bandwidth between racks is often not sufficient if multiple machines in a rack try to communicate significant amounts of data with machines from other racks. Typically, the interconnects of the aggregation switches support higher bandwidths of 10 to 40 Gbps, although interconnects of 100 Gbps are available. Interconnects of even higher capacity can be created by using multiple interconnects in parallel. However, even such high-speed links can be saturated if a large enough number of servers in a rack attempt to communicate at their full connection bandwidth to servers in other racks.

  To avoid the bandwidth bottleneck of a tree structure, data centers typically connect each top-of-rack (edge) switch to multiple aggregation switches. Each aggregation switch in turn is linked to a number of core switches at the next layer. Such an interconnection topology is called a **tree-like topology**; Figure 20.4e shows a tree-like topology with three tiers. The tree-like topology is also referred to as a **fat-tree topology**, although originally the fat-tree topology referred to a tree topology where edges higher in the tree have a higher bandwidth than edges lower in the tree.

  The benefit of the tree-like architecture is that each top-of-rack switch can route its messages through any of the aggregation switches that it is connected to, increasing the inter-rack bandwidth greatly as compared to the tree topology. Similarly, each aggregation switch can communicate with another aggregation switch via any of the core switches that it is connected to, increasing the bandwidth available between the aggregation switches. Further, even if an aggregation or edge switch fails, there are alternative paths through other switches. With appropriate

Telegram: @uni_k

www.konkur.in

routing algorithms, the network can continue functioning even if a switch fails, making the network *fault-tolerant*, at least to failures of one or a few switches.

A tree-like architecture with three tiers can handle a **cluster** of tens of thousands of machines. Although a tree-like topology improves the inter-rack bandwidth greatly compared to a tree topology, parallel processing applications, including parallel storage and parallel database systems, perform best if they are designed in a way that reduces inter-rack traffic.

The tree-like topology and variants of it are widely used in data centers today. The complex interconnection networks in a data center are referred to as a **data center fabric**.

While network topologies are very important for scalability, a key to network performance is network technology used for individual links. The popular technologies include:

- **Ethernet**: The dominant technology for network connections today is the Ethernet technology. Ethernet standards have evolved over time, and the predominant versions used today are 1-gigabit Ethernet and 10-gigabit Ethernet, which support bandwidths of 1 and 10 gigabits per second respectively. Forty-gigabit Ethernet and 100-gigabit Ethernet technologies are also available at a higher cost and are seeing increasing usage. Ethernet protocols can be used over cheaper copper cables for short distances, and over optical fiber for longer distances.
- **Fiber channel**: The Fiber Channel Protocol standard was designed for high-speed interconnection between storage systems and computers, and it is predominantly used to implement storage area networks (described in Section 20.4.6). The different versions of the standard have supported increasing bandwidth over the years, with 16 gigabits per second available as of 2011, and 32 and 128 gigabits per second supported from 2016.
- **Infiniband**: The Infiniband standard was designed for interconnections with a data center; it was specifically designed for high-performance computing applications which need not just very high bandwidth, but also very low latency. The Infiniband standard has evolved, with link speeds of 8-gigabits per second available by 2007 and 24-gigabits per second available by 2014. Multiple links can be aggregated to give a bandwidth of 120 to 290-gigabits per second.

  The latency associated with message delivery is as important as bandwidth for many applications. A key benefit of Infiniband is that it supports latencies as low as 0.7 to 0.5 microseconds. In contrast, Ethernet latencies can be up to hundreds of microseconds in an unoptimized local-area network, while latency-optimized Ethernet implementations still have latencies of several microseconds.

One of the important techniques used to reduce latency is to allow applications to send and receive messages by directly interfacing with the hardware, bypassing the operating system. With the standard implementations of the networking stack, applications

Telegram: @uni_k

www.konkur.in

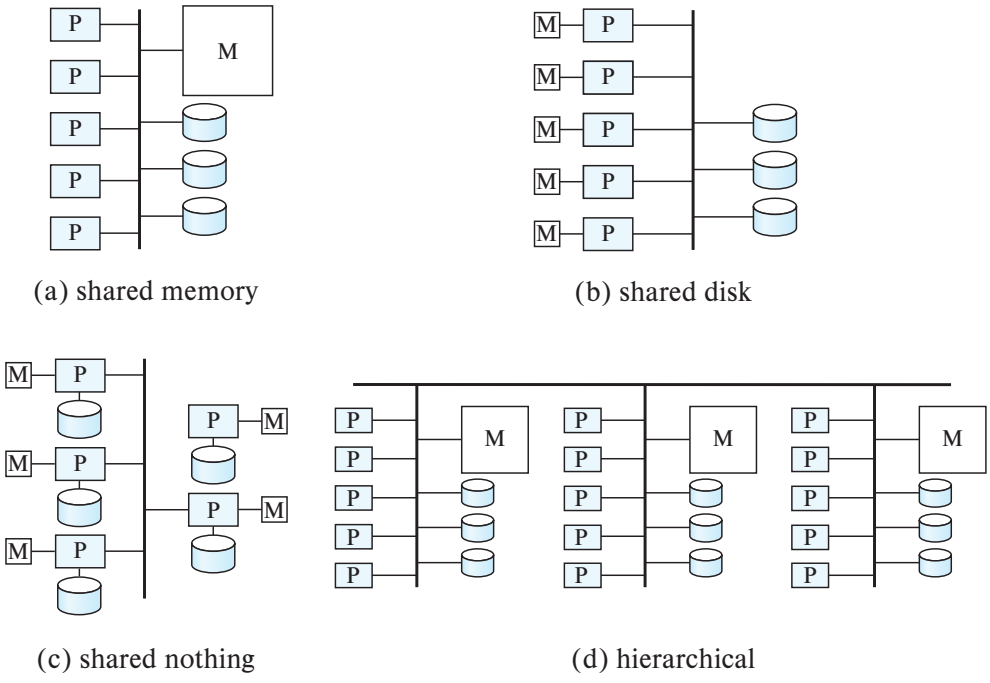


Figure 20.5 Parallel database architectures.

send messages to the operating system, which in turn interfaces with the hardware, which in turn delivers the message to the other computer, where again the hardware interfaces with the operating system, which then interfaces with the application to deliver the message. Support for direct access to the network interface, bypassing the operating system, reduces the communication latency significantly.

Another approach to reducing latency is to use **remote direct memory access** (**RDMA**), a technology which allows a process on one node to directly read or write to memory on another node, without explicit message passing. Hardware support ensures that RDMA can transfer data at very high rates with very low latency. RDMA implementations can use Infiniband, Ethernet, or other networking technologies for physical communication between nodes.

20.4.4 Parallel Database Architectures

There are several architectural models for parallel machines. Among the most prominent ones are those in Figure 20.5 (in the figure, M denotes memory, P denotes a processor, and disks are shown as cylinders):

- **Shared memory**. All the processors share a common memory (Figure 20.5a).

Telegram: @uni_k

www.konkur.in

- **Shared disk**. A set of nodes that share a common set of disks; each node has its own processor and memory (Figure 20.5b). Shared-disk systems are sometimes called **clusters**.
- **Shared nothing**. A set of nodes that share neither a common memory nor common disk (Figure 20.5c).
- **Hierarchical**. This model is a hybrid of the preceding three architectures (Figure 20.5d). This model is the most widely used model today.

In Section 20.4.5 through Section 20.4.8, we elaborate on each of these models.

Note that the interconnection networks are shown in an abstract manner in Figure 20.5. Do not interpret the interconnection networks shown in the figures as necessarily being a bus; in fact other interconnection networks are used in practice. For example, mesh networks are used within a processor, and tree-like networks are often used to interconnect nodes.

20.4.5 Shared Memory

In a **shared-memory** architecture, the processors have access to a common memory, typically through an interconnection network. Disks are also shared by the processors. The benefit of shared memory is extremely efficient communication between processes—data in shared memory can be accessed by any process without being moved with software. A process can send messages to other processes much faster by using memory writes (which usually take less than a microsecond) than by sending a message through a communication mechanism.

Multicore processors with 4 to 8 cores are now common not just in desktop computers, but even in mobile phones. High-end processing systems such as Intel's Xeon processor have up to 28 cores per CPU, with up to 8 CPUs on a board, while the Xeon Phi coprocessor systems contain around 72 cores, as of 2018, and these numbers have been increasing steadily. The reason for the increasing number of cores is that the sizes of features such as logic gates in integrated circuits has been decreasing steadily, allowing more gates to be packed in a single chip. The number of transistors that can be accommodated on a given area of silicon has been doubling approximately every 1 1/2 to 2 years.[4]

Since the number of gates required for a processor core has not increased correspondingly, it makes sense to have multiple processors on a single chip. To maintain a distinction between on-chip multiprocessors and traditional processors, the term **core** is used for an on-chip processor. Thus, we say that a machine has a multicore processor.

[4]Gordon Moore, cofounder of Intel, predicted such an exponential growth in the number of transistors back in the 1960s; his prediction is popularly known as **Moore's law**, even though, technically, it is not a *law*, but rather an observation and a prediction. In earlier decades, processor speeds also increased along with the decrease in the feature sizes, but that trend ended in the mid-2000s since processor clock frequencies beyond a few gigahertz could not be attained without unreasonable increase in power consumption and heat generation. Moores's law is sometimes erroneously interpreted to have predicted exponential increases in processor speeds.

Telegram: @uni_k

www.konkur.in

All the cores on a single processor typically access a shared memory. Further, a system can have multiple processors which can share memory. Another effect of the increasing number of gates has been the steady increase in the size of main memory as well as a decrease in cost, per-byte, of main memory.

Given the availability of multicore processors at a low cost, as well as the concurrent availability of very large amounts of memory at a low cost, shared-memory parallel processing has become increasingly important in recent years.

20.4.5.1 Shared-Memory Architectures

In earlier generation architectures, processors were connected to memory via a bus, with all processor cores and memory banks sharing a single bus. A downside of shared-memory accessed via a common bus is that the bus or the interconnection network becomes a bottleneck, since it is shared by all processors. Adding more processors does not help after a point, since the processors will spend most of their time waiting for their turn on the bus to access memory.

As a result, modern shared-memory architectures associate memory directly with processors; each processor has locally connected memory, which can be accessed very quickly; however, each processor can also access memory associated with other processors; a fast interprocessor communication network ensures that data are fetched with relatively low overhead. Since there is a difference in memory access speed depending on which part of memory is accessed, such an architecture is often referred to as **non-uniform memory architecture** (**NUMA**).

Figure 20.6 shows a conceptual architecture of a modern shared-memory system with multiple processors; note that each processor has a bank of memory directly connected to it, and the processors are linked by a fast interconnect system; processors are also connected to I/O controllers which interface with external storage.

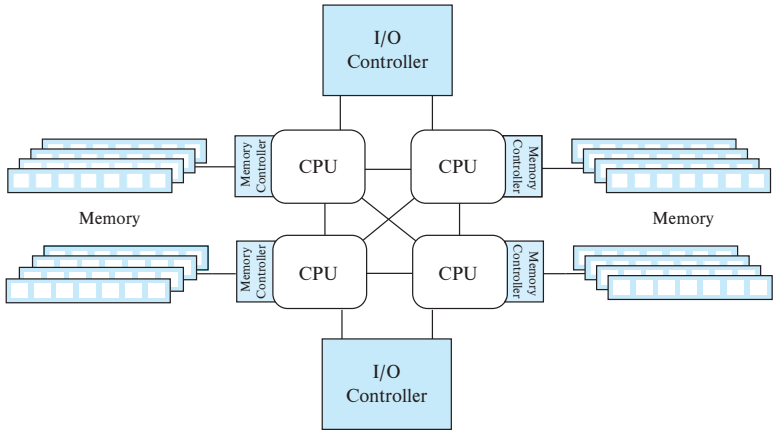


Figure 20.6 Architecture of a modern shared-memory system.

Telegram: @uni_k

www.konkur.in

Because shared-memory architectures require specialized high-speed interconnects between cores and between processors, the number of cores/processors that can be interconnected in a shared-memory system is relatively small. As a result, the scalability of shared-memory parallelism is limited to at most a few hundred cores.

Processor architectures include cache memory, since access to cache memory is much faster than access to main memory (cache can be accessed in a few nanoseconds compared to nearly a hundred nanoseconds for main memory). Large cache memory is particularly important in shared-memory architectures, since a large cache can help minimize the number of accesses to shared memory.

If an instruction needs to access a data item that is not in cache, it must be fetched from main memory. Because main memory is much slower than processors, a significant amount of potential processing speed may be lost while a core waits for data from main memory. These waits are referred to as **cache misses**.

Many processor architectures support a feature called **hyper-threading**, or **hardware threads**, where a single physical core appears as two or more logical cores or threads. Different processes could be mapped to different logical cores. Only one of the logical cores corresponding to a single physical core can actually execute at any time. But the motivation for logical cores is that if the code running on one logical core blocks on a cache miss, waiting for data to be fetched from memory, the hardware of the physical core can start execution of one of the other logical cores instead of idling while waiting for data to be fetched from memory.

A typical multicore processor has multiple levels of cache, with the L1 cache being fastest to access, but also the smallest; lower cache levels such as L2 and L3 are slower (although still much faster than main memory) but considerably larger than the L1 cache. Lower cache levels are usually shared between multiple cores on a single processor. In the cache architecture shown in Figure 20.7, the L1 and L2 caches are local to each of the 4 cores, while the L3 cache is shared by all cores of the processor. data are read into, or written from, cache in units of a **cache line**, which typically consists of 64 consecutive bytes.

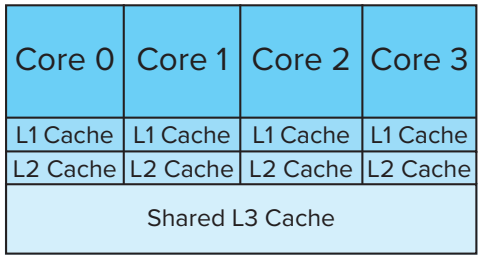


Figure 20.7 Multilevel cache system.

Telegram: @uni_k

www.konkur.in

20.4.5.2 Cache Coherency

Cache coherency is an issue whenever there are multiple cores or processors, each with its own cache. An update done on one core may not be seen by another core, if the local cache on the second core contains an old value of the affected memory location. Thus, whenever an update occurs to a memory location, copies of the content of that memory location that are cached on other caches must be invalidated.

Such invalidation is done lazily in many processor architectures; that is, there may be some time lag between a write to a cache and the dispatch of invalidation messages to other caches; in addition there may be a further lag in processing invalidation messages that are received at a cache. (Requiring immediate invalidation to be done always can cause a significant performance penalty, and thus it is not done in current-generation systems.) Thus, it is quite possible for a write to happen on one processor, and a subsequent read on another processor may not see the updated value.

Such a lack of cache coherency can cause problems if a process expects to see an updated memory location but does not. Modern processors therefore support **memory barrier** instructions, which ensure certain orderings between load/store operations before the barrier and those after the barrier. For example, the **store barrier** instruction (`sfence`) on the x86 architecture forces the processor to wait until invalidation messages are sent to all caches for all updates done prior to the instruction, before any further load/store operations are issued. Similarly, the **load barrier** instruction (`lfence`) ensures all received invalidation messages have been applied before any further load/store operations are issued. The `mfence` instruction does both of these tasks.

Memory barrier instructions must be used with interprocess synchronization protocols to ensure that the protocols execute correctly. Without the use of memory barriers, if the caches are not "strongly" coherent, the following scenario can happen. Consider a situation where a process $P1$ updates memory location A first, then location B; a concurrent process $P2$ running on a different core or processor reads B first and then reads A. With a coherent cache, if $P2$ sees the updated value of B, it must also see the updated value of A. However, in the absence of cache coherence, the writes may be propagated out of order, and $P2$ may thus see the updated value of B but the old value of A. While many architectures disallow out-of-order propagation of writes, there are other subtle errors that can occur due to lack of cache coherency. However, executing `sfence` instructions after each of these writes and `lfence` before each of the reads will always ensure that reads see a cache coherent state. As a result, in the above example, the updated value of B will be seen only if the updated value of A is seen.

It is worth noting that programs do not need to include any extra code to deal with cache coherency, as long as they acquire locks before accessing data, and release locks only after performing updates, since lock acquire and release functions typically include the required memory barrier instructions. Specifically, an `sfence` instruction is executed as part of the lock release code, before the data item is actually unlocked. Similarly an `lfence` is executed right after locking a data item, as part of the lock acquisition

Telegram: @uni_k

www.konkur.in

function, and is thus executed before the item is read. Thus, the reader is guaranteed to see the most recent value written to the data item.

Synchronization primitives supported in a variety of languages also internally execute memory barrier instructions; as a result, programmers who use these primitives need not be concerned about lack of cache coherency.

It is also interesting to note that many processor architectures use a form of hardware-level shared and exclusive locking of memory locations to ensure cache coherency. A widely used protocol, called the MESI protocol, can be understood as follows: Locking is done at the level of cache lines, containing multiple memory locations, instead of supporting locks on individual memory locations, since cache lines are the units of cache access. Locking is implemented in the hardware, rather than in software, to provide the required high performance.

The MESI protocol keeps track of the state of each cache line, which can be Modified (updated after exclusive locking), Exclusive locked (locked but not yet modified, or already written back to memory), Share locked, or Invalid. A read of a memory location automatically acquires a shared lock on the cache line containing that location, while a memory write gets an exclusive lock on the cache line before performing the write. In contrast to database locks, memory lock requests do not wait; instead they immediately revoke conflicting locks. Thus, an exclusive lock request automatically invalidates all cached copies of the cache line and revokes all shared locks on the cache line. Symmetrically, a shared lock request causes any existing exclusive lock to be revoked and then fetches the latest copy of the memory location into cache.

In principle, it is possible to ensure "strong" cache coherency with such a locking-based cache coherence protocol, making memory barrier instructions redundant. However, many implementations include some optimizations that speed up processing, such as allowing delayed delivery of invalidation messages, at the cost of not guaranteeing cache coherence. As a result, memory barrier instructions are required on many processor architectures to ensure cache coherency.

20.4.6 Shared Disk

In the **shared-disk** model, each node has its own processors and memory, but all nodes can access all disks directly via an interconnection network. There are two advantages of this architecture over a shared-memory architecture. First, a shared-disk system can scale to a larger number of processors than a shared-memory system. Second, it offers a cheap way to provide a degree of **fault tolerance**: If a node fails, the other nodes can take over its tasks, since the database is resident on disks that are accessible from all nodes.

We can make the disk subsystem itself fault tolerant by using a RAID architecture, as described in Chapter 12, allowing the system to function even if individual disks fail. The presence of a large number of storage devices in a RAID system also provides some degree of I/O parallelism.

Telegram: @uni_k

www.konkur.in

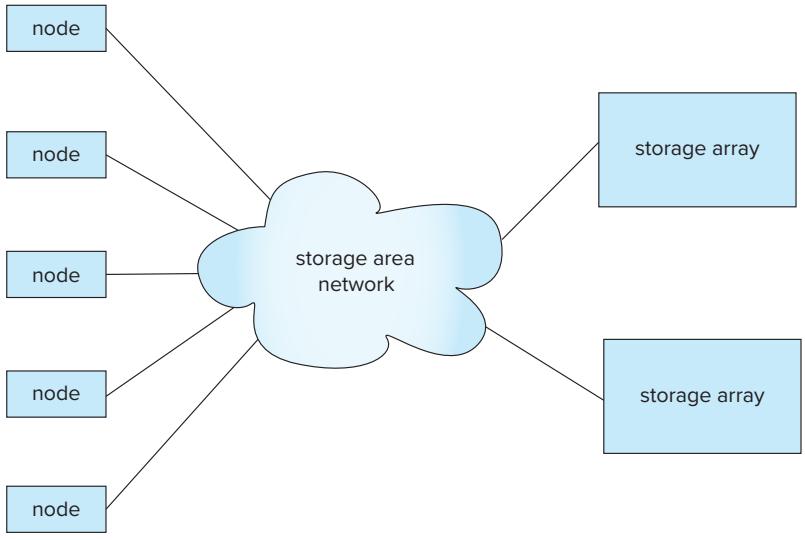


Figure 20.8 Storage-area network.

A **storage-area network** (SAN) is a high-speed local-area network designed to connect large banks of storage devices (disks) to nodes that use the data (see Figure 20.8). The storage devices physically consist of an array of multiple disks but provide a view of a logical disk, or set of disks, that hides the details of the underlying disks. For example, a logical disk may be much larger than any of the physical disks, and a logical disk's size can be increased by adding more physical disks. The processing nodes can access disks as if they are local disks, even though they are physically separate.

Storage-area networks are usually built with redundancy, such as multiple paths between nodes, so if a component such as a link or a connection to the network fails, the network continues to function.

Storage-area networks are well suited for building shared-disk systems. The shared-disk architecture with storage-area networks has found acceptance in applications that do not need a very high degree of parallelism but do require high availability.

Compared to shared-memory systems, shared-disk systems can scale to a larger number of processors, but communication across nodes is slower (up to a few milliseconds in the absence of special-purpose hardware for communication), since it has to go through a communication network.

One limitation of shared-disk systems is that the bandwidth of the network connection to storage in a shared-disk system is usually less than the bandwidth available to access local storage. Thus, storage access can become a bottleneck, limiting scalability.

20.4.7 Shared Nothing

In a **shared-nothing** system, each node consists of a processor, memory, and one or more disks. The nodes communicate by a high-speed interconnection network. A node

Telegram: @uni_k

www.konkur.in

functions as the server for the data on the disk or disks that the node owns. Since local disk references are serviced by local disks at each node, the shared-nothing model overcomes the disadvantage of requiring all I/O to go through a single interconnection network.

Moreover, the interconnection networks for shared-nothing systems, such as the tree-like interconnection network, are usually designed to be scalable, so their transmission capacity increases as more nodes are added. Consequently, shared-nothing architectures are more scalable and can easily support a very large number of nodes.

The main drawbacks of shared-nothing systems are the costs of communication and of nonlocal disk access, which are higher than in a shared-memory or shared-disk architecture since sending data involves software interaction at both ends.

Due to their high scalability, shared-nothing architectures are widely used to deal with very large data volumes, supporting scalability to thousands of nodes, or in extreme cases, even to tens of thousands of nodes.

20.4.8 Hierarchical

The **hierarchical architecture** combines the characteristics of shared-memory, shared-disk, and shared-nothing architectures. At the top level, the system consists of nodes that are connected by an interconnection network and do not share disks or memory with one another. Thus, the top level is a shared-nothing architecture. Each node of the system could actually be a shared-memory system with a few processors. Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system. Thus, a system could be built as a hierarchy, with shared-memory architecture with a few processors at the base, and a shared-nothing architecture at the top, with possibly a shared-disk architecture in the middle. Figure 20.5d illustrates a hierarchical architecture with shared-memory nodes connected together in a shared-nothing architecture.

Parallel database systems today typically run on a hierarchical architecture, where each node supports shared-memory parallelism, with multiple nodes interconnected in a shared-nothing manner.

20.5 Distributed Systems

In a **distributed database system**, the database is stored on nodes located at geographically separated **sites**. The nodes in a distributed system communicate with one another through various communication media, such as high-speed private networks or the internet. They do not share main memory or disks. The general structure of a distributed system appears in Figure 20.9.

The main differences between shared-nothing parallel databases and distributed databases include the following:

Telegram: @uni_k

www.konkur.in

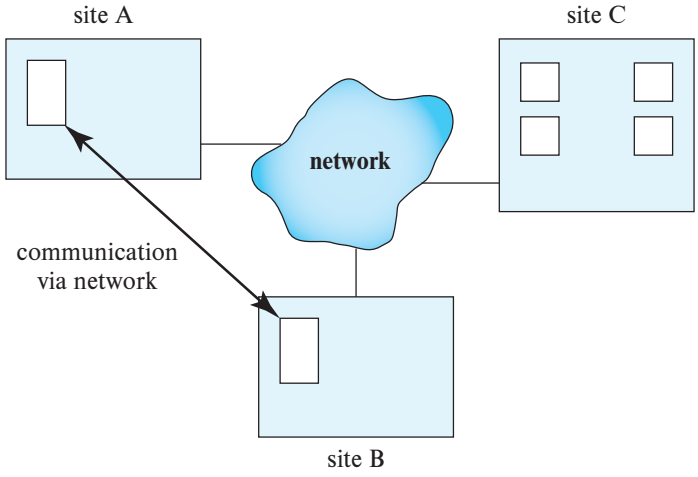


Figure 20.9 A distributed system.

- Distributed databases have sites that are geographically separated. As a result, the network connections have lower bandwidth, higher latency, and greater probability of failures, as compared to networks within a single data center.
  Systems built on distributed databases therefore need to be aware of network latency, and failures, as well as of physical data location. We discuss these issues later in this section. In particular, it is often desirable to keep a copy of the data at a data center close to the end user.
- Parallel database systems address the problem of node failure. However, some failures, particularly those due to earthquakes, fires, or other natural disasters, may affect an entire data center, causing failure of a large number of nodes. Distributed database systems need to continue working even in the event of failure of an entire data center, to ensure high availability. This requires replication of data across geographically separated data centers, to ensure that a common natural disaster does not affect all the data centers. Replication and other techniques to ensure high availability are similar in both parallel and distributed databases, although implementation details may differ.
- Distributed databases may be separately administered, with each site retaining some degree of autonomy of operation. Such databases are often the result of the integration of existing databases to allow queries and transactions to cross database boundaries. However, distributed databases that are built for providing geographic distribution, versus those built by integrating existing databases, may be centrally administered.
- Nodes in a distributed database tend to vary more in size and function, whereas parallel databases tend to have nodes that are of similar capacity.

Telegram: @uni_k

www.konkur.in

- In a distributed database system, we differentiate between local and global transactions. A **local transaction** is one that accesses data only from nodes where the transaction was initiated. A **global transaction**, on the other hand, is one that either accesses data in a node different from the one at which the transaction was initiated, or accesses data in several different nodes.

Web-scale applications today run on data management systems that combine support for parallelism and distribution. Parallelism is used within a data center to handle high loads, while distribution across data centers is used to ensure high availability even in the event of natural disasters. At the lower end of functionality, such systems may be distributed data storage systems that support only limited functionality such as storage and retrieval of data by key, and they may not support schemas, query languages, or transactions; all such higher-level functionality has to be managed by the applications. At the higher end of functionality, there are distributed database systems that support schemas, query language, and transactions. However, one characteristic of such systems is that they are centrally administered.

In contrast, distributed databases that are built by integrating existing database systems have somewhat different characteristics.

- **Sharing data.** The major advantage in building a distributed database system is the provision of an environment where users at one site may be able to access the data residing at other sites. For instance, in a distributed university system, where each campus stores data related to that campus, it is possible for a user in one campus to access data in another campus. Without this capability, the transfer of student records from one campus to another campus would have to rely on some external mechanism.
- **Autonomy.** The primary advantage of sharing data by means of data distribution is that each site can retain a degree of control over data that are stored locally. In a centralized system, the database administrator of the central site controls the database. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to the local database administrator for each site. Depending on the design of the distributed database system, each administrator may have a different degree of **local autonomy**.

In a **homogeneous distributed database** system, nodes share a common global schema (although some relations may be stored only at some nodes), all nodes run the same distributed database-management software, and the nodes actively cooperate in processing transactions and queries.

However, in many cases a distributed database has to be constructed by linking together multiple already-existing database systems, each with its own schema and possibly running different database-management software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in query and transaction processing. Such systems are sometimes called **federated database systems** or **heterogeneous distributed database systems**.

Telegram: @uni_k

www.konkur.in

Nodes in a distributed database communicate over *wide-area networks* (WAN). Although wide-area networks have bandwidth much greater than local-area networks, the bandwidth is usually shared by multiple users/applications and is expensive relative to local-area network bandwidth. Thus, applications that communicate across wide-area networks usually have a lower bandwidth.

Communication in a WAN must also contend with significant **latency**: a message may take up to a few hundred milliseconds to be delivered across the world, both due to speed-of-light delays, and due to queuing delays at a number of routers in the path of the message. Latency in a wide-area setting is a fundamental problem that cannot be reduced beyond a point. Thus, applications whose data and computing resources are distributed geographically have to be carefully designed to ensure that latency does not affect system performance excessively.

Wide-area networks also have to contend with network-link failures, a problem that is relatively rare in local-area networks. In particular, network-link failures may result in two sites that are both alive having no way to communicate with each other, a situation referred to as a **network partition**.[5] In the event of a partition, it may not be possible for a user or an application to access required data. Thus, network partitioning affects the **availability** of a system. Tradeoffs between availability and consistency of data in the event of network partitions are discussed in Section 23.4.

20.6 Transaction Processing in Parallel and Distributed Systems

Atomicity of transactions is an important issue in building a parallel and distributed database system. If a transaction runs across two nodes, unless the system designers are careful, it may commit at one node and abort at another, leading to an inconsistent state. Transaction commit protocols ensure such a situation cannot arise. The *two-phase commit protocol* (2PC) is the most widely used of these protocols.

The 2PC protocol is described in detail in Section 23.2.1, but the key ideas are as follows: The basic idea behind 2PC is for each node to execute the transaction until it enters the partially committed state, and then leave the commit decision to a single coordinator node; the transaction is said to be in the *ready* state at a node at this point. The coordinator decides to commit the transaction only if the transaction reaches the ready state at every node where it executed; otherwise (e.g., if the transaction aborts at any node), the coordinator decides to abort the transaction. Every node where the transaction executed must follow the decision of the coordinator. If a node fails when a transaction is in ready state, when the node recovers from failure it should be in a position to either commit or abort the transaction, depending on the decision of the coordinator.

[5] Do not confuse the term *network partitioning* with the term *data partitioning*; data partitioning refers to dividing up of data items into partitions, which may be stored at different nodes.

Telegram: @uni_k

www.konkur.in

Concurrency control is another issue in parallel and distributed databases. Since a transaction may access data items at several nodes, transaction managers at several nodes may need to coordinate to implement concurrency control. If locking is used, locking can be performed locally at the nodes containing accessed data items, but there is also a possibility of deadlock involving transactions originating at multiple nodes. Therefore deadlock detection needs to be carried out across multiple nodes. Failures are more common in distributed systems since not only may computers fail, but communication links may also fail. Replication of data items, which is the key to the continued functioning of distributed databases when failures occur, further complicates concurrency control. We describe concurrency-control techniques for distributed databases in Section 23.3 (which describes techniques based on locking) and Section 23.3.4 (which describes techniques based on timestamps).

The standard transaction models, based on multiple actions carried out by a single program unit, are often inappropriate for carrying out tasks that cross the boundaries of databases that cannot or will not cooperate to implement protocols such as 2PC. Alternative approaches, based on *persistent messaging* for communication, are generally used for such tasks; persistent messaging is discussed in Section 23.2.3.

When the tasks to be carried out are complex, involving multiple databases and/or multiple interactions with humans, coordination of the tasks and ensuring transaction properties for the tasks become more complicated. *Workflow management systems* are systems designed to help with carrying out such tasks.

20.7 Cloud-Based Services

Traditionally, enterprises purchased and ran servers that execute the database as well as the applications. There is a high cost to maintaining servers, including setting up server room infrastructure dealing with all kinds of failures such as air conditioning and power failures, not to mention failures of CPUs, disks, and other components of the servers. Further, if there is a sudden increase in demand, it is very difficult to add infrastructure to service the demand, and if demand falls, the infrastructure may lie idle.

In contrast, in the **cloud computing** model, applications of an enterprise are executed on an infrastructure that is managed by another company, typically at a data center that hosts a large number of machines used by many different enterprises/users. The service provider may provide not just hardware, but also support platforms such as databases, and application software.

A variety of vendors offer cloud services; these include major vendors such as Amazon, Microsoft, IBM, and Google, and a number of smaller vendors. One of the pioneers of cloud services, Amazon, originally built a large computing infrastructure purely for its internal use; then, seeing a business opportunity, it offered computing infrastructure as a service to other users. Cloud services became very popular within just a few years.

Telegram: @uni_k

www.konkur.in

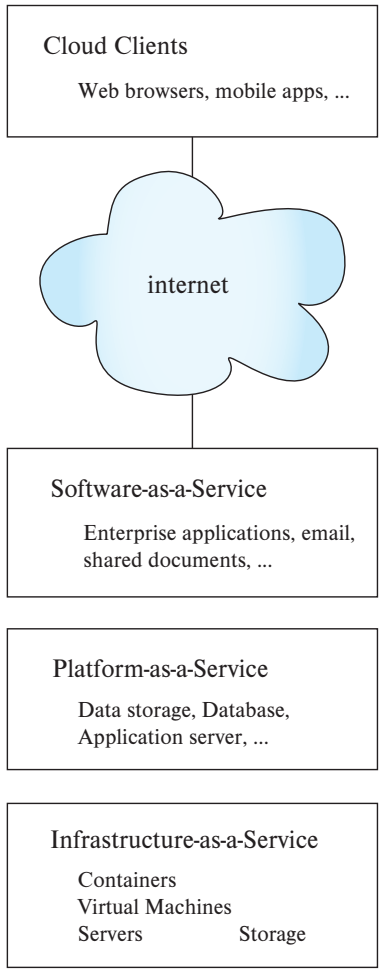


Figure 20.10 Cloud service models.

20.7.1 Cloud Service Models

There are several ways in which cloud computing can be utilized, which are summarized in Figure 20.10. These include infrastructure-as-a-service, platform-as-a-service, and software-as-a-service models.

- In the **infrastructure-as-a-service** model, an enterprise rents computing facilities; for example, an enterprise may rent one or more physical machines, along with disk storage space.

  More frequently, cloud computing providers provide an abstraction of a **virtual machine** (**VM**), which appears to the user to be a real machine. These machines are not "real" machines, but rather are simulated by software that allows a single

Telegram: @uni_k

www.konkur.in

real computer to simulate several independent computers. *Containers* are a lower cost alternative to VMs and are described later in this section. Multiple VMs can run on a single server machine, and multiple containers can run on a single VM or server.

By running a very large data center with many machines, cloud-service providers can exploit economies of scale and deliver computing power at much lower cost than an enterprise can do using its own infrastructure.

Another major advantage of cloud computing is that the cloud-service provider usually has a large number of machines, with spare capacity, and thus an enterprise can rent more (virtual) machines as needed to meet demand and release them at times of light load. The ability to expand or contract capacity at short notice is often referred to as **elasticity**.

The above benefits of on-demand elastic provisioning of server systems have led to the widespread adoption of infrastructure-as-service platforms, especially by companies that anticipate rapid growth in their computing usage. However, due to the potential security risks of storing data outside the enterprise, the use of cloud computing is still limited in high-security enterprise needs, such as banking.

In the infrastructure-as-service model, the client enterprise runs its own software, including database systems, on virtual machines provided by the cloud-service provider; the client has to install the database system and deal with maintenance issues such as backup and restore.

- In the **platform-as-a-service** model, the service provider not only provides computing infrastructure, but it also deploys and manages platforms, such as data storage, databases, and application servers, that are used by application software. The client has to install and maintain application software, such as *enterprise resource planning* (*ERP*) systems, which run on such platform-provided services as application servers, database services, or data storage services.

  - **Cloud-based data storage** platforms provide a service that applications can use to store and retrieve data. The service provider takes care of provisioning sufficient amount of storage and computing power to support the load on the data storage platform. Such storage systems could support files, which are typically large, ranging in size from a few megabytes to thousands of megabytes, supporting millions of such files. Or such storage systems could support data items, which are typically small, ranging from hundreds of bytes to a few megabytes, but supporting billions of such data items. Such distributed file systems and data storage systems are discussed in Section 21.6 and Section 21.7. Database applications using cloud-based storage may run on the same cloud (i.e., the same set of machines), or on another cloud.

    One of the main attractions of cloud-based storage is that it can be used by paying a fee without worrying about purchasing, maintaining, and managing the computer systems on which such a service runs. Further, if there is an

Telegram: @uni_k

www.konkur.in

increase in demand, the number of servers on which the service runs can be increased by paying a larger fee, without having to actually purchase and deploy more servers. The service provider would of course have to deploy extra servers, but they benefit from economies of scale; the cost of deployment, and especially the time to deployment, are greatly reduced compared to what they would be if the end-users did it on their own.

The fees for cloud-based data storage are typically based on the amount of data stored, and amount of data input to, and the amount of data output from, the data storage system.

- **Database-as-a-service** platforms provide a database that can be accessed and queried by clients. Unlike storage services, database-as-a-service platforms provide database functionality such as querying using SQL or other query languages, which data storage systems do not provide. Early offerings of database-as-a-service only supported databases that run on a single node, although the node itself can have a substantial number of processors, memory, and storage. More recently, parallel database systems are being offered as a service on the cloud.

- In the **software-as-a-service** model, the service provider provides the application software as a service. The client does not need to deal with issues such as software installation or upgrades; these tasks are left to the service provider. The client can directly use interfaces provided by the software-as-a-service provider, such as web interfaces, or mobile app interfaces that provide a front end, with the application software acting as the back end.

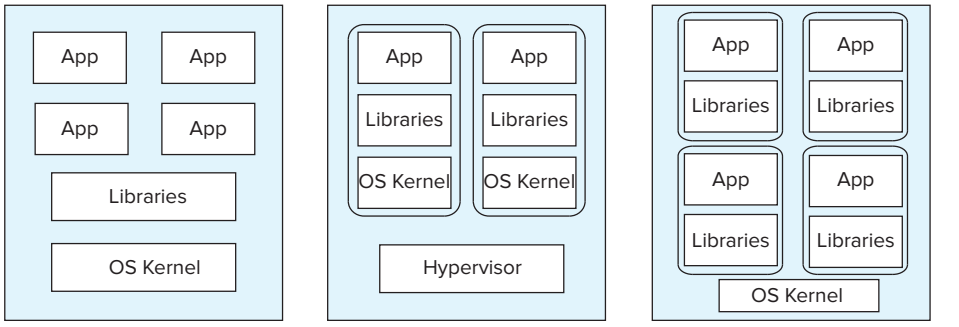
The concept of virtual machines was developed in the 1960s to allow an expensive mainframe computer to be shared concurrently by users running different operating systems. Although computers are now much cheaper, there is still a cost associated with supplying electrical power to the computers and maintaining them; virtual machines allow this cost to be shared by multiple concurrent users. Virtual machines also ease the task of moving services to new machines: a virtual machine can be shut down on one physical server and restarted on another physical server with very little delay or downtime. This feature is particularly important for quick recovery in the event of hardware failure or upgrade.

Although multiple virtual machines can run on a single real machine, each VM has a high overhead, since it runs an entire operating system internally. When a single organization wishes to run a number of services, if it creates a separate VM for each service, the overhead can be very high. If multiple applications are run on one machine (or VM), two problems often arise: (1) applications conflict on network ports by each trying to listen to the same network port, and (2) applications require different versions of shared libraries, causing conflicts.

Containers solve both these problems; applications run in a container, which has its own IP address and its own set of shared libraries. Each application can consist of

Telegram: @uni_k

www.konkur.in



a) Multiple applications on a single machine

b) Each application running on its own VM, with multiple VMs running in a machine

c) Each application running in its own container, with multiple containers running in a machine

Figure 20.11 Application deployment alternatives.

multiple processes, all running within the same container. The cost of using containers to run applications is much less than the alternative of running each application in its own VM, since many containers can share the same operating system kernel. Each container appears to have its own file system, but the files are all stored in a common underlying file system across all containers. Processes within a container can interact with each other through the file system as well as interprocess communication, but they can interact with processes from other containers only via network connections.

Figure 20.11 depicts the different deployment alternatives for a set of applications. Figure 20.11a shows the alternative of multiple applications running in a single machine, sharing libraries and operating-system kernel. Figure 20.11b shows the alternative of running each application in its own VM, with multiple VMs running on a single machine. The different VMs running on a single real machine are managed by a software layer called the *hypervisor*. Figure 20.11c shows the alternative of using containers, with each container having its own libraries, and multiple containers running on a single machine. Since containers have lower overheads, a single machine can support more containers than VMs.

Containers provide low-cost support for elasticity, since more containers can be deployed very quickly on existing virtual machines, instead of starting up fresh virtual machines.

Many applications today are built as a collection of multiple **services**, each of which runs as a separate process, offering a network API; that is, the functions provided by the service are invoked by creating a network connection to the process and sending a service request over the network connection. Such an application architecture, which builds an application as a collection of small services, is called a **microservices architecture**. Containers fit the microservices architecture very well, since they provide a very low overhead mechanism to execute processes supporting each service.

Telegram: @uni_k

www.konkur.in

Docker is a very widely used container platform, while **Kubernetes** is a very popular platform that provides not only containers, but also a microservices platform. Kubernetes allows applications to specify declaratively their container needs, and it automatically deploys and links multiple containers to execute the application. It can also manage a number of pods, which allow multiple containers to share storage (file system) and network (IP address) while allowing containers to retain their own copies of shared libraries. Furthermore, it can manage elasticity by controlling the deployment of additional containers when required. Kubernetes can support application scalability by load-balancing API requests across a collection of containers that all run copies of the same application. Users of the API do not need to know what IP addresses (each corresponding to a container) the service is running on, and they can instead connect to a single IP address. The load balancer distributes the requests from the common IP address to a set of containers (each with its own IP address) running the service.

20.7.2 Benefits and Limitations of Cloud Services

Many enterprises are finding the model of cloud computing and services beneficial. The cloud model saves client enterprises the need to maintain a large system-support staff and allows new enterprises to begin operation without having to make a large, up-front capital investment in computing systems. Further, as the needs of the enterprise grow, more resources (computing and storage) can be added as required; the cloud-computing vendor generally has very large clusters of computers, making it easy for the vendor to allocate resources on demand.

Users of cloud computing must be willing to accept that their data are held by another organization. This may present a variety of risks in terms of security and legal liability. If the cloud vendor suffers a security breach, client data may be divulged, causing the client to face legal challenges from its customers. Yet the client has no direct control over cloud-vendor security. These issues become more complex if the cloud vendor chooses to store data (or replicas of data) in a foreign country. Various legal jurisdictions differ in their privacy laws. So, for example, if a German company's data are replicated on a server in New York, then the privacy laws of the United States may apply instead of or in addition to those of Germany or the European Union. The cloud vendor might be required to release client data to the U.S. government even though the client never knew that its data would be stored in a location under U.S. jurisdiction. Specific cloud vendors offer their clients varying degrees of control over how their data are distributed geographically and replicated.

Despite the drawbacks, the benefits of cloud services are great enough that there is a rapidly growing market for such services.

20.8 Summary

- Centralized database systems run entirely on a single computer. Database systems designed for multiuser systems need to support the full set of transaction features.

Telegram: @uni_k

www.konkur.in

Such systems are usually designed as servers that accept requests from applications via SQL or their own APIs.

- Parallelism with a small number of cores is referred to as coarse-grained parallelism. Parallelism with a large number of processors is referred to as fine-grained parallelism.
- Transaction servers have multiple processes, possibly running on multiple processors. So that these processes have access to common data, such as the database buffer, systems store such data in shared memory. In addition to processes that handle queries, there are system processes that carry out tasks such as lock and log management and checkpointing.
- Access to shared memory is controlled by a mutual-exclusion mechanism based on machine-level atomic instructions (test-and-set or compare-and-swap).
- Data-server systems supply raw data to clients. Such systems strive to minimize communication between clients and servers by caching data and locks at the clients. Parallel database systems use similar optimizations.
- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network. Speedup measures how much we can increase processing speed by increasing parallelism for a single transaction. Scaleup measures how well we can handle an increased number of transactions by increasing parallelism. Interference, skew, and start-up costs act as barriers to getting ideal speedup and scaleup.
- The components of a parallel system are connected via several possible types of interconnection networks: bus, ring, mesh, hypercube, or a tree-like topology.
- Parallel database architectures include the shared-memory, shared-disk, shared-nothing, and hierarchical architectures. These architectures have different trade-offs of scalability versus communication speed.
- Modern shared-memory architectures associate some memory with each processor, resulting in a non-uniform memory architecture (NUMA). Since each processor has its own cache, there is a problem of ensuring cache coherency, that is, consistency of data across the caches of multiple processors.
- Storage-area networks are a special type of local-area network designed to provide fast interconnection between large banks of storage devices and multiple computers.
- A distributed database system is a collection of partially independent database systems that (ideally) share a common schema and coordinate processing of transactions that access nonlocal data.

Telegram: @uni_k

www.konkur.in

- Cloud services may be provided at a variety of levels: The infrastructure-as-a-service model provides clients with a virtual machine on which clients install their own software. The platform-as-a-service model provides data-storage, database, and application servers in addition to virtual machines, but the client needs to install and maintain application software. The software-as-a-service model provides that application software plus the underlying platform.
- Organizations using cloud services have to consider a wide variety of technical, economic, and legal issues in order to ensure the privacy and security of data and adequate performance despite the likelihood of data being stored at a remote location.

Review Terms

- Centralized Database Systems
  - Single-user system
  - Multiuser system
  - Server systems
  - Embedded databases
  - Servers
  - Coarse-grained parallelism
  - Fine-grained parallelism
- Server System Architectures
  - Transaction-server
  - Query-server
  - Data-server systems
  - Server processes
- Mutual exclusion
- Atomic instructions
- Data caching
- Parallel Systems
  - Coarse-grain parallel machine
  - Massively parallel machine
  - Fine-grain parallel machine
  - Data center
- Decision-support queries
- Measure of performance
  - Throughput
  - Response time
  - Linear speedup
  - Sublinear speedup
  - Linear scaleup
  - Sublinear scaleup
- Sequential computation
- Amdahl's law
- Start-up costs
- Interconnection network
  - Bus
  - Ring
  - Mesh
  - Hypercube
  - Tree-like
  - Edge switches
- Aggregation switch

Telegram: @uni_k

www.konkur.in

- Ethernet
- Fiber channel
- Infiniband
- Remote direct memory access (RDMA)
- Parallel Database Architectures
  - Shared memory
  - Shared disk
  - Clusters
  - Shared nothing
  - Hierarchical
- Moore's law
- NUMA
- Cache misses
- Hyper-threading
- Hardware threads
- Cache
- Shared-disk
- Fault tolerance
- Storage-area network (SAN)
- Distributed database system
- Local autonomy
- Homogeneous distributed database
- Federated database systems
- Heterogeneous distributed database systems
- Latency
- Network partition
- Availability
- Cloud computing
- Infrastructure-as-a-service
- Platform-as-a-service
- Cloud-based data storage
- Database-as-a-service
- Software-as-a-service
- Microservices architecture

Practice Exercises

20.1 Is a multiuser system necessarily a parallel system? Why or why not?

20.2 Atomic instructions such as compare-and-swap and test-and-set also execute a memory fence as part of the instruction on many architectures. Explain what is the motivation for executing the memory fence, from the viewpoint of data in shared memory that is protected by a mutex implemented by the atomic instruction. Also explain what a process should do before releasing a mutex.

20.3 Instead of storing shared structures in shared memory, an alternative architecture would be to store them in the local memory of a special process and access the shared data by interprocess communication with the process. What would be the drawback of such an architecture?

20.4 Explain the distinction between a *latch* and a *lock* as used for transactional concurrency control.

20.5 Suppose a transaction is written in C with embedded SQL, and about 80 percent of the time is spent in the SQL code, with the remaining 20 percent spent

Telegram: @uni_k

www.konkur.in

in C code. How much speedup can one hope to attain if parallelism is used only for the SQL code? Explain.

20.6 Consider a pair of processes in a shared memory system such that process *A* updates a data structure, and then sets a flag to indicate that the update is completed. Process *B* monitors the flag, and starts processing the data structure only after it finds the flag is set.

Explain the problems that could arise in a memory architecture where writes may be reordered, and explain how the sfence and lfence instructions can be used to ensure the problem does not occur.

20.7 In a shared-memory architecture, why might the time to access a memory location vary depending on the memory location being accessed?

20.8 Most operating systems for parallel machines (i) allocate memory in a local memory area when a process requests memory, and (ii) avoid moving a process from one core to another. Why are these optimizations important with a NUMA architecture?

20.9 Some database operations such as joins can see a significant difference in speed when data (e.g., one of the relations involved in a join) fits in memory as compared to the situation where the data do not fit in memory. Show how this fact can explain the phenomenon of **superlinear speedup**, where an application sees a speedup greater than the amount of resources allocated to it.

20.10 What is the key distinction between homogeneous and federated distributed database systems?

20.11 Why might a client choose to subscribe only to the basic infrastructure-as-a-service model rather than to the services offered by other cloud service models?

20.12 Why do cloud-computing services support traditional database systems best by using a virtual machine, instead of running directly on the service provider's actual machine, assuming that data is on external storage?

Exercises

20.13 Consider a bank that has a collection of sites, each running a database system. Suppose the only way the databases interact is by electronic transfer of money between themselves, using persistent messaging. Would such a system qualify as a distributed database? Why?

20.14 Assume that a growing enterprise has outgrown its current computer system and is purchasing a new parallel computer. If the growth has resulted in many more transactions per unit time, but the length of individual transactions has

Telegram: @uni_k

www.konkur.in

not changed, what measure is most relevant—speedup, batch scaleup, or transaction scaleup? Why?

20.15 Database systems are typically implemented as a set of processes (or threads) accessing shared memory.

a. How is access to the shared-memory area controlled?

b. Is two-phase locking appropriate for serializing access to the data structures in shared memory? Explain your answer.

20.16 Is it wise to allow a user process to access the shared-memory area of a database system? Explain your answer.

20.17 What are the factors that can work against linear scale up in a transaction processing system? Which of the factors are likely to be the most important in each of the following architectures: shared-memory, shared disk, and shared nothing?

20.18 Memory systems today are divided into multiple modules, each of which can be serving a separate request at a given time, in contrast to earlier architectures where there was a single interface to memory. What impact has such a memory architecture have on the number of processors that can be supported in a shared-memory system?

20.19 Assume we have data items $d_1, d_2, \ldots, d_n$ with each d_i protected by a lock stored in memory location M_i.

a. Describe the implementation of lock-X(d_i) and unlock(d_i) via the use of the test-and-set instruction.

b. Describe the implementation of lock-X(d_i) and unlock(d_i) via the use of the compare-and-swap instruction.

20.20 In a shared-nothing system data access from a remote node can be done by remote procedure calls, or by sending messages. But remote direct memory access (RDMA) provides a much faster mechanism for such data access. Explain why.

20.21 Suppose that a major database vendor offers its database system (e.g., Oracle, SQL Server DB2) as a cloud service. Where would this fit among the cloud-service models? Why?

20.22 If an enterprise uses its own ERP application on a cloud service under the platform-as-a-service model, what restrictions would there be on when that enterprise may upgrade the ERP system to a new version?

Telegram: @uni_k

www.konkur.in

Further Reading

[Hennessy et al. (2017)] provides an excellent introduction to the area of computer architecture, including the topics of shared-memory architectures and cache coherency, parallel computing architectures, and cloud computing, which we covered in this chapter. [Gray and Reuter (1993)] provides the classic textbook description of transaction processing, including the architecture of client–server and distributed systems. [Ozsu and Valduriez (2010)] provides textbook coverage of distributed database systems. [Abts and Felderman (2012)] provides an overview of data center networking.

Bibliography

[Abts and Felderman (2012)] D. Abts and B. Felderman, "A Guided Tour of Datacenter Networking", *Communications of the ACM*, Volume 55, Number 6 (2012), pages 44–51.

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Hennessy et al. (2017)] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*, 6th edition, Morgan Kaufmann (2017).

[Ozsu and Valduriez (2010)] T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, 3nd edition, Prentice Hall (2010).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 21



Parallel and Distributed Storage

As we discussed in Chapter 20, parallelism is used to provide speedup, where queries are executed faster because more resources, such as processors and disks, are provided. Parallelism is also used to provide scaleup, where increasing workloads are handled without increased response time, via an increase in the degree of parallelism.

In this chapter, we discuss techniques for data storage and indexing in parallel database systems.

21.1 Overview

We first describe, in Section 21.2 and Section 21.3, how to partition data amongst multiple nodes. We then discuss, in Section 21.4, replication of data and parallel indexing (in Section 21.5). Our description focuses on shared-nothing parallel database systems, but the techniques we describe are also applicable to distributed database systems, where data are stored in a geographically distributed manner.

File systems that run on a large number of nodes, called distributed file systems, are a widely used way to store data in a parallel system. We discuss distributed file systems in Section 21.6.

In recent years parallel data storage and indexing techniques have been extensively used for storage of nonrelational data, including unstructured text data, and semi-structured data in XML, JSON, or other formats. Such data are often stored in parallel **key-value stores**, which store data items with an associated key. The techniques we describe for parallel storage of relational data can also be used for key-value stores which are discussed in Section 21.7. We use the term **data storage system** to refer to both key-value stores, and the data storage and access layer of parallel database systems.

Query processing in parallel and distributed databases is discussed in Chapter 22, while and transaction processing in parallel and distributed databases is discussed in Chapter 23.

Telegram: @uni_k

www.konkur.in

21.2 Data Partitioning

In its simplest form, **I/O parallelism** refers to reducing the time required to retrieve data from disk by partitioning the data over multiple disks.[1]

At the lowest level, RAID systems allow blocks to be partitioned across multiple disks, allowing them to be accessed in parallel. Blocks are usually allocated to different disks in a round-robin fashion, as we saw in Section 12.5. For example, if there are n disks numbered 0 to $n - 1$, round-robin allocation assigns block i to disk i mod n. However, the block-level partitioning techniques supported by RAID systems do not offer any control in terms of which tuples of a relation are stored on which disk or node. Therefore, parallel database systems typically do not use block-level partitioning and instead perform partitioning at the level of tuples.

In systems with multiple nodes (computers), each with multiple disks, partitioning can potentially be specified to the level of individual disks. However, parallel database systems typically focus on partitioning data across nodes and leave it to the operating system on each node to decide on assigning blocks to disks within the node.

In a parallel storage system, the tuples of a relation are partitioned (divided) among many nodes, so that each tuple resides on one node; such partitioning is referred to as **horizontal partitioning**. Several partitioning strategies have been proposed for horizontal partitioning, which we study next.

We note that *vertical partitioning*, discussed in Section 13.6 in the context of columnar storage, is orthogonal to horizontal partitioning. (As an example of vertical partitioning, a relation $r(A, B, C, D)$ where A is a primary key, may be vertically partitioned into $r(A, B)$ and $r(A, C, D)$, if many queries require B values, while C and D values are large in size and not required for many queries.) Once tuples are horizontally partitioned, they may be stored in a vertically partitioned manner at each node.

We also note that several database vendors use the term *partitioning* to denote the partitioning of tuples of a relation r into multiple physical relations $r_1, r_2, \ldots, r_n$, where all the physical relations r_i are stored in a single node. The relation r is not stored, but treated as a view defined by the query $r_1 \cup r_2 \cup \ldots \cup r_n$. Such *intra-node partitioning* of a relation is typically used to ensure that frequently accessed tuples are stored separately from infrequently accessed tuples and is different from horizontal partitioning across nodes. Intra-node partitioning is described in more detail in Section 25.1.4.3.

In the rest of this chapter, as well as in subsequent chapters, we use the term *partitioning* to refer to *horizontal partitioning* across multiple nodes.

21.2.1 Partitioning Strategies

We present three basic *data-partitioning strategies* for partitioning tuples. Assume that there are n nodes, $N_1, N_1, \ldots, N_n$, across which the data are to be partitioned.

[1] As in earlier chapters, we use the term *disk* to refer to persistent storage devices, such as magnetic hard disks and solid-state drives.

Telegram: @uni_k

www.konkur.in

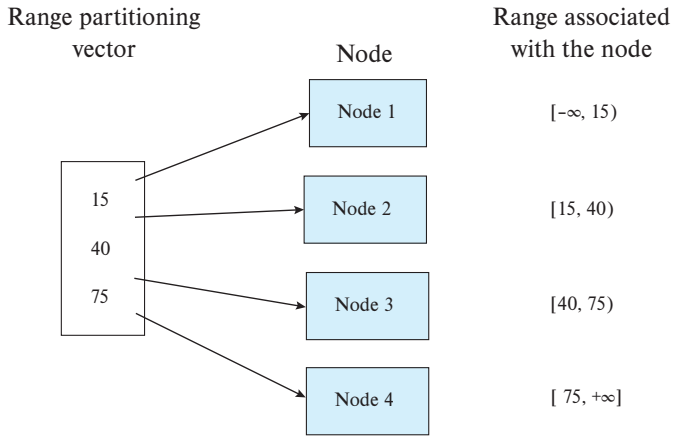


Figure 21.1 Example of range partitioning vector.

- **Round-robin**. This strategy scans the relation in any order and sends the ith tuple fetched during the scan to node number $N_{((i-1) \bmod n)+1}$. The round-robin scheme ensures an even distribution of tuples across nodes; that is, each node has approximately the same number of tuples as the others.
- **Hash partitioning**. This declustering strategy designates one or more attributes from the given relation's schema as the partitioning attributes. A hash function is chosen whose range is $\{1, 2, \ldots, n\}$. Each tuple of the original relation is hashed on the partitioning attributes. If the hash function returns i, then the tuple is placed on node N_i.[2]
- **Range partitioning**. This strategy distributes tuples by assigning contiguous attribute-value ranges to each node. It chooses a partitioning attribute, A, and a **partitioning vector** $[v_1, v_2, \ldots, v_{n-1}]$, such that, if $i < j$, then $v_i < v_j$. The relation is partitioned as follows: Consider a tuple t such that $t[A] = x$. If $x < v_1$, then t goes on node N_1. If $x \geq v_{n-1}$, then t goes on node N_n. If $v_i \leq x < v_{i+1}$, then t goes on node N_{i+1}.

Figure 21.1 shows an example of a range partitioning vector. In the example in the figure, values less than 15 are mapped to Node 1. Values in the range $[15, 40)$, i.e., values ≥ 15 but < 40. are mapped to Node 2; Values in the range $[40, 75)$, i.e., values ≥ 40 but < 75, are mapped to Node 3, while values > 75 are mapped to Node 4.

We now consider how partitioning is maintained when a relation is updated.

1. When a tuple is inserted into a relation, it is sent to the appropriate node based on the partitioning strategy.

[2]Hash-function design is discussed in Section 24.5.1.1.

Telegram: @uni_k

www.konkur.in

2. If a tuple is deleted, its location is first found based on the value of its partitioning attribute (for round-robin, all partitions are searched). The tuple is then deleted from wherever it is located.
3. If a tuple is updated, its location is not affected if either round-robin partitioning is used or if the update does not affect a partitioning attribute.
   However, if range partitioning or hash partitioning is used, and the update affects a partitioning attribute, the location of the tuple may be affected. In this case:
   a. The original tuple is deleted from the original location, and
   b. The updated tuple is inserted and sent to the appropriate node based on the partitioning strategy used.

21.2.2 Comparison of Partitioning Techniques

Once a relation has been partitioned among several nodes, we can retrieve it in parallel, using all the nodes. Similarly, when a relation is being partitioned, it can be written to multiple nodes in parallel. Thus, the transfer rates for reading or writing an entire relation are much faster with I/O parallelism than without it. However, reading an entire relation, or *scanning a relation*, is only one kind of access to data. Access to data can be classified as follows:

1. Scanning the entire relation.
2. Locating a tuple associatively (e.g., *employee_name* = "Campbell"); these queries, called **point queries**, seek tuples that have a specified value for a specific attribute.
3. Locating all tuples for which the value of a given attribute lies within a specified range (e.g., 10000 < *salary* < 20000); these queries are called **range queries**.

The different partitioning techniques support these types of access at different levels of efficiency:

- **Round-robin.** The scheme is ideally suited for applications that wish to read the entire relation sequentially for each query. With this scheme, both point queries and range queries are complicated to process, since each of the n nodes must be used for the search.
- **Hash partitioning.** This scheme is best suited for point queries based on the partitioning attribute. For example, if a relation is partitioned on the *telephone_number* attribute, then we can answer the query "Find the record of the employee with *telephone_number* = 555-3333" by applying the partitioning hash function to 555-3333 and then searching that node. Directing a query to a single node saves the start-up cost of initiating a query on multiple nodes and leaves the other nodes free to process other queries.

Telegram: @uni_k

www.konkur.in

Hash partitioning is also useful for sequential scans of the entire relation. If the hash function is a good randomizing function, and the partitioning attributes form a key of the relation, then the number of tuples in each of the nodes is approximately the same, without much variance. Hence, the time taken to scan the relation is approximately $1/n$ of the time required to scan the relation in a single node system.

The scheme, however, is not well suited for point queries on nonpartitioning attributes. Hash-based partitioning is also not well suited for answering range queries, since, typically, hash functions do not preserve proximity within a range. Therefore, all the nodes need to be scanned for range queries to be answered.

- **Range partitioning.** This scheme is well suited for point and range queries on the partitioning attribute. For point queries, we can consult the partitioning vector to locate the node where the tuple resides. For range queries, we consult the partitioning vector to find the range of nodes on which the tuples may reside. In both cases, the search narrows to exactly those nodes that might have any tuples of interest.

  An advantage of this feature is that, if there are only a few tuples in the queried range, then the query is typically sent to one node, as opposed to all the nodes. Since other nodes can be used to answer other queries, range partitioning results in higher throughput while maintaining good response time. On the other hand, if there are many tuples in the queried range (as there are when the queried range is a larger fraction of the domain of the relation), many tuples have to be retrieved from a few nodes, resulting in an I/O bottleneck (hot spot) at those nodes. In this example of **execution skew**, all processing occurs in one—or only a few—partitions. In contrast, hash partitioning and round-robin partitioning would engage all the nodes for such queries, giving a faster response time for approximately the same throughput.

The type of partitioning also affects other relational operations, such as joins, as we shall see in Section 22.3 and Section 22.4.1.

Thus, the choice of partitioning technique also depends on the operations that need to be executed. In general, hash partitioning or range partitioning are preferred to round-robin partitioning.

Partitioning is important for large relations. Large databases that benefit from parallel storage often have some small relations. Partitioning is not a good idea for such small relations, since each node would end up with just a few tuples. Partitioning is worthwhile only if each node would contain at least a few disk blocks worth of data. Small relations are best left unpartitioned, while medium-sized relations could be partitioned across some of the nodes, rather than across all the nodes, in a large system.

21.3 Dealing with Skew in Partitioning

When a relation is partitioned (by a technique other than round-robin), there may be a skew in the distribution of tuples, with a high percentage of tuples placed in some

Telegram: @uni_k

www.konkur.in

partitions and fewer tuples in other partitions. Such an imbalance in the distribution of data is called **data distribution skew**. Data distribution skew may be caused by one of two factors.

- **Attribute-value skew**, which refers to the fact that some values appear in the partitioning attributes of many tuples. All the tuples with the same value for the partitioning attribute end up in the same partition, resulting in skew.
- **Partition skew**, which refers to the fact that there may be load imbalance in the partitioning, even when there is no attribute skew.

Attribute-value skew can result in skewed partitioning regardless of whether range partitioning or hash partitioning is used. If the partition vector is not chosen carefully, range partitioning may result in partition skew. Partition skew is less likely with hash partitioning if a good hash function is chosen.

As Section 20.4.2 noted, even a small skew can result in a significant decrease in performance. Skew becomes an increasing problem with a higher degree of parallelism. For example, if a relation of 1000 tuples is divided into 10 parts, and the division is skewed, then there may be some partitions of size less than 100 and some partitions of size more than 100; if even one partition happens to be of size 200, the speedup that we would obtain by accessing the partitions in parallel is only 5, instead of the 10 for which we would have hoped. If the same relation has to be partitioned into 100 parts, a partition will have 10 tuples on an average. If even one partition has 40 tuples (which is possible, given the large number of partitions) the speedup that we would obtain by accessing them in parallel would be 25, rather than 100. Thus, we see that the loss of speedup due to skew increases with parallelism.

In addition to skew in the distribution of tuples, there may be **execution skew** even if there is no skew in the distribution of tuples, if queries tend to access some partitions more often than others. For example, suppose a relation is partitioned by the timestamp of the tuples, and most queries refer to recent tuples; then, even if all partitions contain the same number of tuples, the partition containing recent tuples would experience a significantly higher load.

In the rest of this section, we consider several approaches to handling skew.

21.3.1 Balanced Range-Partitioning Vectors

Data distribution skew in range partitioning can be avoided by choosing a **balanced range-partitioning vector**, which evenly distributes tuples across all nodes.

A balanced range-partitioning vector can be constructed by sorting, as follows: The relation is first sorted on the partitioning attributes. The relation is then scanned in sorted order. After every $1/n$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector. Here, n denotes the number of partitions to be constructed.

Telegram: @uni_k

www.konkur.in

The main disadvantage of this method is the extra I/O overhead incurred in doing the initial sort. The I/O overhead for constructing balanced range-partitioning vectors can be reduced by using a precomputed frequency table, or **histogram**, of the attribute values for each attribute of each relation. Figure 21.2 shows an example of a histogram for an integer-valued attribute that takes values in the range 1 to 25. It is straightforward to construct a balanced range-partitioning function given a histogram on the partitioning attributes. A histogram takes up only a little space, so histograms on several different attributes can be stored in the system catalog. If the histogram is not stored, it can be computed approximately by sampling the relation, using only tuples from a randomly chosen subset of the disk blocks of the relation. Using a random sample allows the histogram to be constructed in much less time than it would take to sort the relation.

The preceding approach for creating range-partitioning vectors addresses data-distribution skew; extensions to handle execution skew are left as an exercise for the reader (Exercise 21.3).

A drawback of the above approach is that it is *static*: the partitioning is decided at some point and is not automatically updated as tuples are inserted, deleted, or updated. The partitioning vectors can be recomputed, and the data repartitioned, whenever the system detects skew in data distribution. However, the cost of repartitioning can be quite large, and doing it periodically would introduce a high load which can affect normal processing. Dynamic techniques for avoiding skew, which can adapt in a continuous and less disruptive fashion, are discussed in Section 21.3.2 and in Section 21.3.3.

21.3.2 Virtual Node Partitioning

Another approach to minimizing the effect of skew is to use *virtual nodes*. In the **virtual nodes** approach, we pretend there are several times as many *virtual nodes* as the number

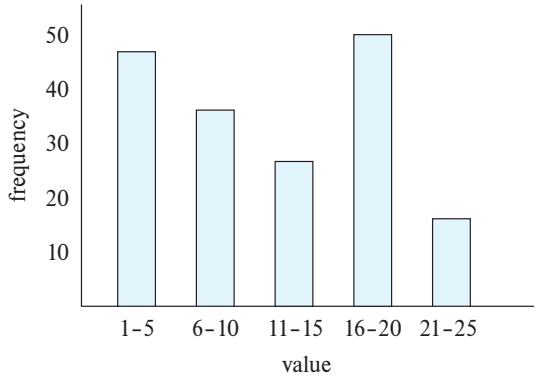


Figure 21.2 Example of histogram.

Telegram: @uni_k

www.konkur.in

of real nodes. Any of the partitioning techniques described earlier can be used, but to map tuples and work to virtual nodes instead of to real nodes.[3]

Virtual nodes, in turn, are mapped to real nodes. One way to map virtual nodes to real nodes is round-robin allocation; thus, if there are n real nodes numbered 1 to n, virtual node i is mapped to real node $((i-1) mod n) + 1$. The idea is that even if one range had many more tuples than the others because of skew, these tuples would get split across multiple virtual nodes ranges. Round-robin allocation of virtual nodes to real nodes would distribute the extra work among multiple real nodes, so that one node does not have to bear all the burden.

A more sophisticated way of doing the mapping is by tracking the number of tuples in each virtual node, and the load (e.g., the number of accesses per second) on each virtual node. Virtual nodes are then mapped to real nodes in a way that balances the number of stored tuples as well as the load across the real nodes. Thus, data-distribution skew and execution skew can be minimized.

The system must then record this mapping and use it to route accesses to the correct real node. If virtual nodes are numbered by consecutive integers, this mapping can be stored as an array *virtual_to_real_map[]*, with m entries, where there are m virtual nodes; the ith element of this array stores the real node to which virtual node i is mapped.

Yet another benefit of the virtual node approach is that it allows **elasticity of storage**, that is, as the load on the system increases it is possible to add more resources (nodes) to the system to handle the load. When a new node is added, some of the virtual nodes are *migrated* to the new real node, which can be done without affecting any of the other virtual nodes. If the amount of data mapped to each virtual node is small, the migration of a virtual node from one node to another can be done relatively fast, minimizing disruption.

21.3.3 Dynamic Repartitioning

While the virtual-node approach can reduce skew with range partitioning as well as hash partitioning, it does not work very well if the data distribution changes over time, resulting in some virtual nodes having a very large number of tuples, or a very high execution load. For example, if partitioning was done by timestamps of records, the last timestamp range would get an increasing number of records, as more records are inserted, while other ranges would not get any new records. Thus, even if the initial partitioning is balanced, it could become increasingly skewed over time.

Skew can be dealt with by recomputing the partitioning scheme entirely. However, repartitioning the data based on the new partitioning scheme would, in general, be a very expensive operation. In the preceding example, we would end up moving a significant number of records from each partition to a partition that precedes it in timestamp

[3]The virtual node approach is also called the **virtual processor** approach, a term used in earlier editions of this book; since the *term virtual* processor is now commonly used in a different sense in the context of virtual machines, we now use the term *virtual node*.

Telegram: @uni_k

www.konkur.in

order. When dealing with large amounts of data, such repartitioning would be unreasonably expensive.

Dynamic repartitioning can be done in an efficient manner by instead exploiting the virtual node scheme. The basic idea is to split a virtual node into two virtual nodes when it has too many tuples, or too much load; the idea is very similar to a B^+-tree node being split into two nodes when it is overfull. One of the newly created virtual nodes can then be migrated to a different node to rebalance the data stored at each node, or the load at each node.

Considering the preceding example, if the virtual node corresponding to a range of timestamps 2017-01-01 to *MaxDate* were to become overfull, the partition could be split into two partitions. For example, if half the tuples in this range have timestamps less than 2018-01-01, one partition would have timestamps from 2017-01-01 to less than 2018-01-01, and the other would have tuples with timestamps from 2018-01-01 to *MaxDate*. To rebalance the number of tuples in a real node, we would just need to move one of the virtual nodes to a new real node.

Dynamic repartitioning in this way is very widely used in parallel databases and parallel data storage systems today. In data storage systems, the term **table** refers to a collection of data items. Tables are partitioned into multiple **tablets**. The number of tablets into which a table is divided is much larger than the number of real nodes in the system; thus tablets correspond to virtual nodes.

The system needs to maintain a **partition table**, which provides a mapping from the partitioning key ranges to a tablet identifier, as well as the real node on which the tablet data reside. Figure 21.3 shows an example of a partition table, where the partition key is a date. Tablet0 stores records with key value $<$ 2012-01-01. Tablet1 stores records with key values $\geq$ 2012-01-01, but $<$ 2013-01-01. Tablet2 stores records with key values $\geq$ 2013-01-01, but $<$ 2014-01-01, and so on. Finally, Tablet6 stores values $\geq$ 2017-01-01.

Read requests must specify a value for the partitioning attribute, which is used to identify the tablet which could contain a record with that key value; a request that does not specify a value for the partitioning attribute would have to be sent to all tablets. A read request is processed by using the partitioning key value v to identify the tablet

2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node1
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
MaxDate	Tablet6	Node1

Figure 21.3 Example of a partition table.

Telegram: @uni_k

www.konkur.in

whose range of keys contains v, and then sending the request to the real node where the tablet resides. The request can be handled efficiently at that node by maintaining, for each tablet, an index on the partitioning key attribute.

Write, insert, and delete requests are processed similarly, by routing the requests to the correct tablet and real node, using the mechanism described above for reads.

The above scheme allows tablets to be split if they become too big; the key range corresponding to the tablet is split into two, with a newly created tablet getting half the key range. Records whose key range is mapped to the new tablet are then moved from the original tablet to the new tablet. The partition table is updated to reflect the split, so requests are then correctly directed to the appropriate tablet.

If a real node gets overloaded, either due to a large number of requests or due to too much data at the node, some of the tablets from the node can be moved to a different real node that has a lower load. Tablets can also be moved similarly in case one of the real nodes has a large amount of data, while another real node has less data. Finally, if a new real node joins a system, some tables can be moved from existing nodes to the new node. Whenever a tablet is moved to a different real node, the partition table is updated; subsequent requests will then be sent to the correct real node.

Figure 21.4 shows the partition table from Figure 21.3 after Tablet6, which had values $\geq$ 2017-01-01, has been split into two: Tablet6 now has values $\geq$ 2017-01-01, but $<$ 2018-01-01, while the new tablet, Tablet7, has values $\geq$ 2018-01-01. Such a split could be caused by a large number of inserts into Tablet6, making it very large; the split rebalances the sizes of the tablets.

Note also that Tablet1, which was in Node1, has now been moved to Node0 in Figure 21.4. Such a tablet move could be because Node1 is overloaded due to excessive data, or due to a high number of requests.

Most parallel data storage systems store the partition table at a **master** node. However, to support a large number of requests each second, the partition table is usually replicated, either to all client nodes that access data or to multiple **routers**. Routers accept read/write requests from clients and forward the requests to the appropriate

2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node0
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
2018-01-01	Tablet6	Node1
MaxDate	Tablet7	Node1

Figure 21.4 Example partition table after tablet split and tablet move.

Telegram: @uni_k

www.konkur.in

real node containing the tablet/virtual nodes based on the key values specified in the request.

An alternative fully distributed approach is supported by a hash based partitioning scheme called **consistent hashing**. In the consistent hashing approach, keys are hashed to a large space, such as 32 bit integers. Further, node (or virtual node) identifiers are also hashed to the same space. A key k_i could be logically mapped to the node n_j whose hash value $h(n_j)$ is the highest value among all nodes satisfying $h(n_j) < h(k_i)$. But to ensure that every key is assigned to a node, hash values are treated as lying on a cycle similar to the face of a clock, where the maximum hash value *maxhash* is immediately followed by 0. Then, key k_i is then logically mapped to the node n_j whose hash value $h(n_j)$ is the closest among all nodes, when we move anti-clockwise in the circle from $h(k_i)$.

Distributed hash tables based on this idea have been developed where there is no need for either a master node or a router; instead each participating node keeps track of a few other peer nodes, and routing is implemented in a cooperative manner. New nodes can join the system, and integrate themselves by following specified protocols in a completely peer-to-peer manner, without the need for a master. See the Further Reading section at the end of the chapter for references providing further details.

21.4 Replication

With a large number of nodes, the probability that at least one node will malfunction in a parallel system is significantly greater than in a single-node system. A poorly designed parallel system will stop functioning if any node fails. Assuming that the probability of failure of a single node is small, the probability of failure of the system goes up linearly with the number of nodes. For example, if a single node would fail once every 5 years, a system with 100 nodes would have a failure every 18 days.

Parallel data storage systems must, therefore, be resilient to failure of nodes. Not only should data not be lost in the event of a node failure, but also, the system should continue to be *available*, that is, continue to function, even during such a failure.

To ensure tuples are not lost on node failure, tuples are replicated across at least two nodes, and often three nodes. If a node fails, the tuples that it stored can still be accessed from the other nodes where the tuples are replicated.[4]

Tracking the replicas at the level of individual tuples would result in significant overhead in terms of storage and query processing. Instead, replication is done at the level of partitions (tablets, nodes, or virtual nodes). That is, each partition is replicated; the locations of the partition replicas are recorded as part of the partition table.

Figure 21.5 shows a partition table with replication of tablets. Each tablet is replicated in two nodes.

[4]Caching also results in replication of data, but with the aim of speeding up access. Since data may be evicted from cache at any time, caching does not ensure availability in the event

Telegram: @uni_k

www.konkur.in

2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

Figure 21.5 Partition table of Figure 21.4 with replication.

The database system keeps track of failed nodes; requests for data stored at a failed node are automatically routed to the backup nodes that store a replica of the data. Issues of how to handle the case where one or more replicas are stored at a currently failed node are addressed briefly in Section 21.4.2, and in more detail later, in Section 23.4.

21.4.1 Location of Replicas

Replication to two nodes provides protection from data loss/unavailability in the event of single node failure, while replication to three nodes provides protection even in the event of two node failures. If all nodes where a partition is replicated fail, obviously there is no way to prevent data loss/unavailability. Systems that use low-cost commodity machines for data storage typically use three-way replication, while systems that use more reliable machines typically use two-way replication.

There are multiple possible failure modes in a parallel system. A single node could fail due to some internal fault. Further, it is possible for all the nodes in a rack to fail if there is some problem with the rack such as, for example, failure of power supply to the entire rack, or failure of the network switches in a rack, making all the nodes in the rack inaccessible. Further, there is a possibility of failure of an entire data center, for example, due to fire, flooding, or a large-scale power failure.

The location of the nodes where the replicas of a partition are stored must, therefore, be chosen carefully, to maximize the probability of at least one copy being accessible even during a failure. Such replication can be within a data center or across data centers.

- **Replication within a data center**: Since single node failures are the most common failure mode, partitions are often replicated to another node.
  With the tree-like interconnection topology commonly used in data center networks (described in Section 20.4.3) network bandwidth within a rack is much

Telegram: @uni_k

www.konkur.in

higher than the network bandwidth between racks. As a result, replication to another node within the same rack as the first node reduces network demand on the network between racks. But to deal with the possibility of a rack failure, partitions are also replicated to a node on a different rack.

- **Replication across data centers**: To deal with the possibility of failure of an entire data center, partitions may also be replicated at one or more geographically separated data centers. Geographic separation is important to deal with disasters such as earthquakes or storms that may shut down all data centers in a geographic region.

  For many web applications, round-trip delays across a long-distance network can affect performance significantly, a problem that is increasing with the use of Ajax applications that require multiple rounds of communication between the browser and the application. To deal with this problem, users are connected with application servers that are closest to them geographically, and data replication is done in such a way that one of the replicas is in the same data center as (or at least, geographically close to) the application server.

Suppose all the partitions at a node N_1 are replicated at a single node N_2, and N_1 fails. Then, node N_2 will have to handle all the requests that would originally have gone to N_1, as well as requests routed to node N_2. As a result, node N_2 would have to perform twice as much work as other nodes in the system, resulting in execution skew during failure of node N_1.

To avoid this problem, the replicas of partitions residing at a node, say N_1, are spread across multiple other nodes. For example, consider a system with 10 nodes and two-way replication. Suppose node N_1 had one of the replicas of partitions p_1 through p_9. Then, the other replica of partitions p_1 could be stored on N_2, of p_2 on N_3, and so on to p_9 on N_{10}. Then in the event of failure of N_1, nodes N_2 through N_{10} would share the extra work equally, instead of burdening a single node with all the extra work.

21.4.2 Updates and Consistency of Replicas

Since each partition is replicated, updates made to tuples in a partition must be performed on all the replicas of the partition. For data that is never updated after it has been created, reads can be performed at any of the replicas, since all of them will have the same value. If a storage system ensures that all replicas are exclusive-locked and updated atomically (using, for example, the two-phase commit protocol which we will see in Section 23.2.1), reads of a tuple can be performed (after obtaining a shared lock) at any of the replicas and will see the most recent version of the tuple.

If data are updated, and replicas are not updated atomically, different replicas may temporarily have different values. Thus, a read may see a different value depending on which replica it accesses. Most applications require that read requests for a tuple must

Telegram: @uni_k

www.konkur.in

receive the most recent version of the tuple; updates that are based on reading an older version could result in a *lost update problem*.

One way of ensuring that reads get the latest value is to treat one of the replicas of each partition as a **master replica**. All updates are sent to the master replica and are then propagated to other replicas. Reads are also sent to the master replica, so that reads get the latest version of any data item even if updates have not yet been applied to the other replicas.

If a master replica fails, a new master is assigned for that partition. It is important to ensure that every update operation performed by the old master has also been seen by the new master. Further, the old master may have updated some of the replicas, but not all, before it failed; the new master must complete the task of updating all the replicas. We discuss details in Section 23.6.2.

It is important to know which node is the (current) master for each partition. This information can be stored along with the partition table. Specifically, the partition table must record, in addition to the range of key values assigned to that partition, where the replicas of the partition are stored, and further which replica is currently the master.

Three solutions are commonly used to update replicas.

- The *two-phase commit* (2PC) protocol, which ensures that multiple updates performed by a transaction are applied atomically across multiple sites, is described in Section 23.2. This protocol can be used with replicas to ensure that an update is performed atomically on all replicas of a tuple.

  We assume for now that all replicas are available and can be updated. Issues of how to allow two-phase commit to continue execution in the presence of failures, when some replicas may not be reachable, are discussed in Section 23.4.

- Persistent messaging systems, described in Section 23.2.3, which guarantee that a message is delivered once it is sent. Persistent messaging systems can be used to update replicas as follows: An update to a tuple is registered as a persistent message, sent to all replicas of the tuple. Once the message is recorded, the persistent messaging system ensures it will be delivered to all replicas. Thus, all replicas will get the update, eventually; the property is known as **eventual consistency** of replicas.

  However, there may be a delay in message delivery, and during that time some replicas may have applied an update while others have not. To ensure that reads get a consistent version, reads are performed only at a master replica, where updates are made first. (If the site with a master replica of a tuple has failed, another replica can take over as the master replica after ensuring all pending persistent messages with updates have been applied.) Details are presented in Section 23.6.2.

- Protocols called *consensus protocols*, that allow updates of replicas to proceed even in the face of failures, when some replicas may not be reachable, can be used to manage update of replicas. Unlike the preceding protocols, consensus protocols can work even without a designated master replica. We study consensus protocols in Section 23.8.

Telegram: @uni_k

www.konkur.in

21.5 Parallel Indexing

Indices in a parallel data storage system can be divided into two kinds: local and global indices. In the following discussion, when virtual node partitioning is used, the term *node* should be understood to mean virtual node (or equivalently, tablet).

- A **local index** is an index built on tuples stored in a particular node; typically, such an index would be built on all the partitions of a given relation. The index contents are stored on the same node as the data.
- A **global index** is an index built on data stored across multiple nodes; a global index can be used to efficiently find matching tuples, regardless of where the tuples are stored.

  While the contents of a global index could be stored at a single central location, such a scheme would result in poor scalability; as a result, the contents of a global index should partitioned across multiple nodes.

A **global primary index** on a relation is a global index on the attributes on which the tuples of the relation are partitioned. A global index on partitioning attribute K is constructed by merely creating local indices on K on each partition.

A query that is intended to retrieve tuples with a specific key value $k1$ for K can be answered by first finding which partition could hold the key value $k1$, and then using the local index in that partition to find the required tuples.

For example, suppose the *student* relation is partitioned on the attribute ID, and a global index is to be constructed on the attribute ID. All that is required is to construct a local index on ID on each partition. Figure 21.6(a) shows a global primary index on the *student* relation, on attribute ID; the local indices are not shown explicitly in the figure.

A query that is intended to retrieve tuples with a specific value for ID, say 557, can be answered by first using the partitioning function on ID to first find which partition could contain the specified ID value 557; the query is then sent to the corresponding node, which uses the local index on ID to locate the required tuples.

Note that ID is the primary key for the relation *student*; however, the above scheme would work even if the partitioning attribute were not the primary key. The scheme can be extended to handle range queries, provided the partitioning function is itself based on ranges of values; a partitioning scheme based on hashing cannot support range queries.

A **global secondary index** on a relation is a global index whose index attributes do not match the attributes on which the tuples are partitioned.

Suppose the partitioning attributes are K_p, while the index attributes are K_i, with $K_p \neq K_i$. One approach for answering a selection query on attributes K_i is as follows: If a local index is created on K_i on each partition of the relation, the query is sent to each

Telegram: @uni_k

www.konkur.in

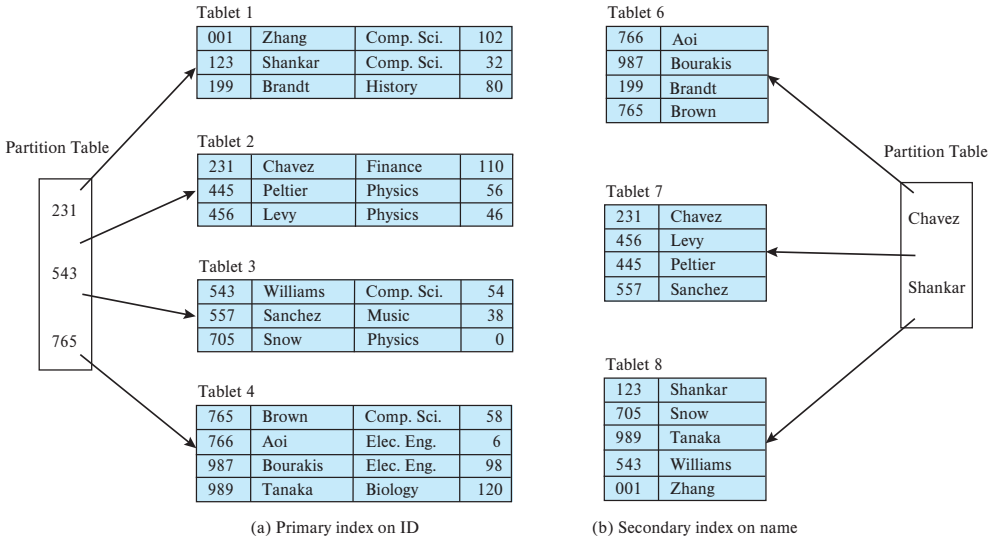


Figure 21.6 Global primary and secondary indices on *student* relation

partition, and the local index is used to find matching tuples. Such an approach using local indices is very inefficient if the number of partitions is large, since every partition has to be queried, even if only one or a few partitions contain matching tuples.

We now illustrate an efficient scheme for constructing a global secondary index by using an example. Consider again the *student* relation partitioned on the attribute ID, and suppose a global index is to be constructed on the attribute *name*. A simple way to construct such an index is to create a set of (*name*, ID) tuples, with one tuple per *student* tuple; let us call this set of tuples *index_name*. Now, the *index_name* tuples are partitioned on the attribute *name*. A local index on *name* is then constructed on each partition of *index_name*. In addition, a global index is created on the ID, which is the partitioning attribute. Figure 21.6(b) shows a secondary index on the *student* relation on attribute *name*; local indices are not explicitly shown in the figure.

Now, a query that needs to retrieve students with a given name can be handled by first examining the partition function of *index_name* to find which partition could store *index_name* tuples with the given name; the query is then sent to that partition, which uses the local index on *name* to find the corresponding ID value. Next, the global index on the ID value is used to find the required tuple.

Note that in the example above, the partitioning attribute ID does not have duplicates; hence it suffices to add only the index key *name* and the attribute ID to the *index_name* relation. Otherwise, further attributes would have to be added to ensure tuples can be uniquely identified, as described next.

In general, given a relation r, which is partitioned on a set of attributes K_p, if we wish to create a global secondary index on a set of attributes K_i, we create a new relation r_i^s, containing the following attributes:

Telegram: @uni_k

www.konkur.in

1. K_i, and K_p
2. If the partitioning attributes K_p have duplicates, we would have to add further attributes K_u, such that $K_p \cup K_u$ is a key for the relation being indexed.

The relation r_i^s is partitioned on K_i, and a local index is created on K_i. In addition, a local index on attributes (K_p, K_u) is created on each partition of relation r.

We now consider how to use a global secondary index to answer a query. Consider a query that specifies a particular value v for K_i. The query is processed as follows:

1. The relevant partition of r_i^s for the value v is found using the partitioning function on K_i.
2. Use the local index on K_i at the above partition to find tuples of r_i^s that have the specified value v for K_i.
3. The tuples in the preceding result are partitioned based on the K_p value and sent to the corresponding nodes.
4. At each node, the tuples received from the preceding step are used along with the local index on r on attributes $K_p \cup K_u$, to find matching r tuples.

Note that relation r_i^s is basically a materialized view defined as $r_i^s = \Pi_{K_i, K_p, K_u}(r)$. Whenever r is modified by inserts, deletions, or updates to tuples, the materialized view r_i^s must be correspondingly updated.

Note also that updates to a tuple of r at some node may result in updates to tuples of r_i^s at other nodes. For example, in Figure 21.6, if the name of ID 001 is updated from Zhang to Yang, the tuple (Zhang, 001) at Tablet8 will be updated to (Yang, 001); since both Zhang and Yang belong in the same partition of the secondary index, no other partition is affected. On the other hand, if the name is updated from Zhang to Bolin, the tuple (Zhang, 001) will be deleted from Tablet8 and a new entry (Bolin, 001) added to Tablet6.

Performing the above updates to the secondary index as part of the same transaction that updates the base relation requires updates to be committed atomically across multiple nodes. Two-phase commit, discussed in Section 23.2, can be used for this task. Alternatives based on persistent messaging can also be used as described in Section 23.2.3, provided it is acceptable for the secondary index to be somewhat out of date.

21.6 Distributed File Systems

A **distributed file system** stores files across a large collection of machines while giving a single-file-system view to clients. As with any file system, there is a system of file names and directories, which clients can use to identify and access files. Clients do not need to bother about where the files are stored.

Telegram: @uni_k

www.konkur.in

The goal of first-generation distributed file systems was to allow client machines to access files stored on one or more file servers. In contrast, later-generation distributed file systems, which we focus on, address distribution of file blocks across a very large number of nodes. Such distributed file systems can store very large amounts of data and support very large numbers of concurrent clients. A landmark system in this context was the Google File System (GFS), developed in the early 2000s, which saw widespread use within Google. The open-source Hadoop File System (HDFS) is based on the GFS architecture and is now very widely used.

Distributed file systems are generally designed to efficiently store large files whose sizes range from tens of megabytes to hundreds of gigabytes or more. However, they are designed to store moderate numbers of such files, of the order of millions; they are typically not designed to stores billions of different files. In contrast, the parallel data storage systems we have seen earlier are designed to store very large numbers (billions or more) of data items, whose size can range from small (tens of bytes) to medium (a few megabytes).

As in parallel data storage systems, the data in a distributed file system are stored across a number of nodes. Since files can be much larger than data items in a data storage system, files are broken up into multiple blocks. The blocks of a single file can be partitioned across multiple machines. Further, each file block is replicated across multiple (typically three) machines, so that a machine failure does not result in the file becoming inaccessible.

File systems typically support two kinds of *metadata*:

1. A directory system, which allows a hierarchical organization of files into directories and subdirectories, and
2. A mapping from a file name to the sequence of identifiers of blocks that store the actual data in each file.

In the case of a centralized file system, the block identifiers help locate blocks in a storage device such as a disk. In the case of a distributed file system, in addition to providing a block identifier, the file system must provide the location (node identifier) where the block is stored; in fact, due to replication, the file system provides a set of node identifiers along with each block identifier.

In the rest of this section, we describe the organization of the Hadoop File System (HDFS), which is shown in Figure 21.7; the architecture of HDFS is derived from that of the Google File System (GFS). The nodes (machines) which store data blocks in HDFS are called **datanodes**. Blocks have an associated ID, and datanodes map the block ID to a location in their local file system where the block is stored.

The file system metadata too can be partitioned across many nodes, but unless carefully architected, this could lead to bad performance. GFS and HDFS took a simpler and more pragmatic approach of storing the file system metadata at a single node, called the **namenode** in HDFS.

Telegram: @uni_k

www.konkur.in

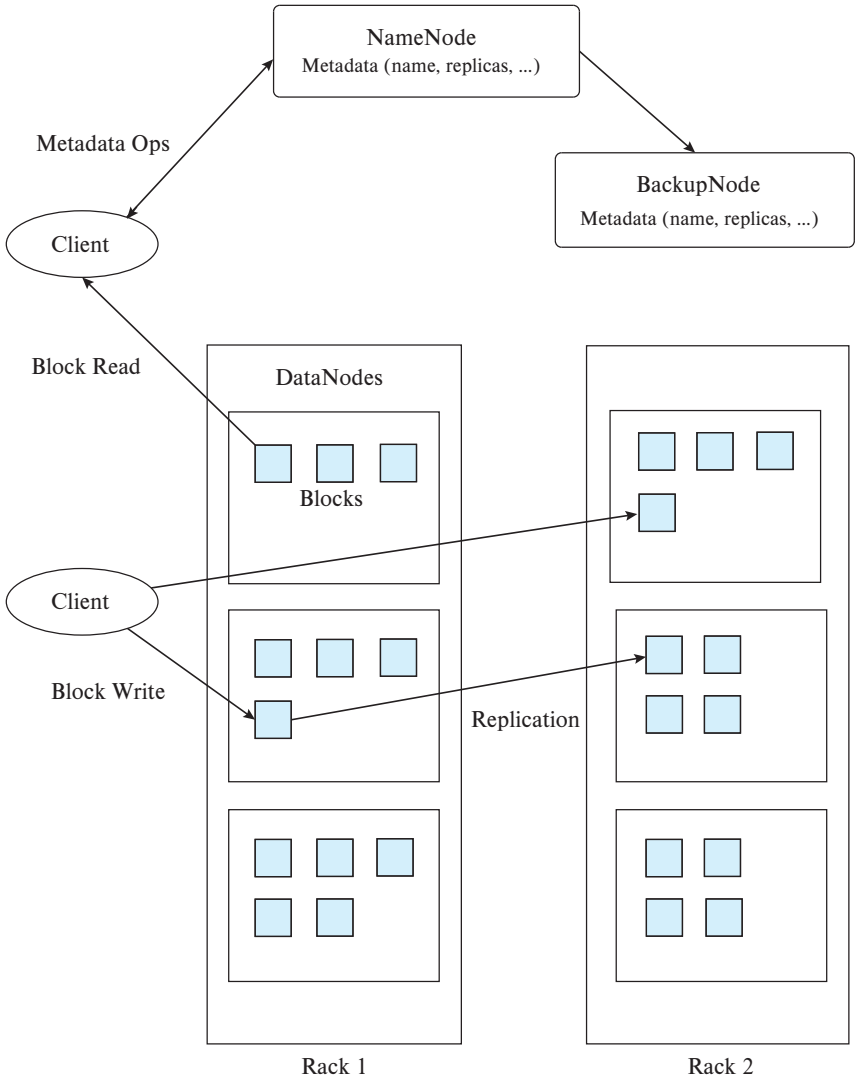


Figure 21.7 Hadoop Distributed File System (HDFS) architecture

Since all metadata reads have to go to the namenode, if a disk access were required to satisfy a metadata read, the number of requests that could be satisfied per second would be very small. To ensure acceptable performance, HDFS namenodes cache the entire metadata in memory; the size of memory then becomes a limiting factor in the number of files and blocks that the file system can manage. To reduce the memory size, HDFS uses very large block sizes (typically 64 MB) to reduce the number of blocks that the namenode must track for each file. Despite this, the limited amount of main memory on most machines constrains namenodes to support only a limited number of

Telegram: @uni_k

www.konkur.in

files (of the order of millions). However, with main-memory sizes of many gigabytes, and block sizes of tens of megabytes, an HDFS system can comfortably handle many petabytes of data.

In any system with a large number of datanodes, datanode failures are a frequent occurrence. To deal with datanode failures, data blocks must be replicated to multiple datanodes. If a datanode fails, the block can still be read from one of the other datanodes that stores a replica of the block. Replication to three datanodes is widely used to provide high availability, without paying too high a storage overhead.

We now consider how a file open and read request is satisfied with HDFS. First, the client contacts the namenode, with the name of the file. The namenode finds the list of IDs of blocks containing the file data and returns to the client the list of block IDs, along with the set of nodes that contain replicas of each of the blocks. The client then contacts any one of the replicas for each block of the file, sending it the ID of the block, to retrieve the block. In case that particular replica does not respond, the client can contact any of the other replicas.

To satisfy a write request, the client first contacts the namenode, which allocates blocks, and decides which datanodes should store replicas of each block. The metadata are recorded at the namenode and sent back to the client. The client then writes the block to all the replicas. As an optimization to reduce network traffic, HDFS implementations may choose to store two replicas in the same rack; in that case, the block write is performed to one replica, which then copies the data to the second replica on the same rack. When all the replicas have processed the write of a block, an acknowledgment is sent to the client.

Replication introduces the problem of consistency of data across the replicas in case the file is updated. As an example, suppose one of the replicas of a data block is updated, but due to system failure, another replica does not get updated; then the system could end up with inconsistent states across the replicas. And what value is read would depend on which replica is accessed, which is not an acceptable situation.

While data storage systems in general need to deal with consistency, using techniques that we study in Chapter 23, some distributed file systems such as HDFS take a different approach: namely, not allowing updates. In other words, a file can be appended to, but data that are written cannot be updated. As each block of the file is written, the block is copied to the replicas. The file cannot be read until it is *closed*, that is, all data have been written to the file, and the blocks have been written successfully at all their replicas. The model of writing data to a file is sometimes referred to as **write-once-read-many** access model. Others such as GFS allow updates and detect certain inconsistent states caused by failures while writing to replicas; however, transactional (atomic) updates are not supported.

The restriction that files cannot be updated, but can only be appended to, is not a problem for many applications of a distributed file system. Applications that require updates should use a data-storage system that supports updates instead of using a distributed file system.

Telegram: @uni_k

www.konkur.in

21.7 Parallel Key-Value Stores

Many Web applications need to store very large numbers (many billions) of relatively small records (of size ranging from a few kilobytes to a few megabytes). Storage would have to be distributed across thousands of nodes. Storing such records as files in a distributed file system is infeasible, since file systems are not designed to store such large numbers of small files. Ideally, a massively parallel relational database should be used to store such data. But the parallel relational databases available in the early 2000s were not designed to work at a massive scale; nor did they support the ability to easily add more nodes to the system without causing significant disruption to ongoing activities.

A number of parallel key-value storage systems were developed to meet the needs of such web applications. A **key-value store** provides a way to store or update a data item (value) with an associated key and to retrieve the data item with a given key. Some key-value stores treat the data items as uninterpreted sequences of bytes, while others allow a schema to be associated with the data item. If the system supports definition of a schema for data items, it is possible for the system to create and maintain secondary indices on specified attributes of data items.

Key-value stores support two very basic functions on tables: `put(table, key, value)`, used to store values, with an associated key, in a table, and `get(table, key)`, which retrieves the stored value associated with the specified key. In addition, they may support other functions, such as range queries on key values, using `get(table, key1, key2)`.

Further, many key-value stores support some form of flexible schema.

- Some allow column names to be specified as part of a schema definition, similar to relational data stores.
- Others allow columns to be added to, or deleted from, individual tuples; such key-value stores are sometimes referred to as **wide-column stores**. Such key-value stores support functions such as `put(table, key, columnname, value)`, to store a value in a specific column of a row identified by the key (creating the column if it does not already exist), and `get(table, key, columnname)`, which retrieves the value for a specific column of a specific row identified by the key. Further, `delete(table, key, columnname)` deletes a specific column from a row.
- Yet other key-value stores allow the value stored with a key to have a complex structure, typically based on JSON; they are sometimes referred to as **document stores**.

The ability to specify a (partial) schema of the stored value allows the key-value store to evaluate selection predicates at the data store; some stores also use the schema to support secondary indices.

We use the term *key-value store* to include all the above types of data stores; however, some people use the term *key-value store* to refer more specifically to those that

Telegram: @uni_k

www.konkur.in

do not support any form of schema and treat the value as an uninterpreted sequence of bytes.

Parallel key-value stores typically support *elasticity*, whereby the number of nodes can be increased or decreased incrementally, depending on demand. As nodes are added, tablets can be moved to the new nodes. To reduce the number of nodes, tablets can be moved away from some nodes, which can then be removed from the system.

Widely used parallel key-value stores that support flexible columns (also known as wide-column stores) include Bigtable from Google, Apache HBase, Apache Cassandra (originally developed at Facebook), and Microsoft Azure Table Storage from Microsoft, among others. Key-value stores that support a schema include Megastore and Spanner from Google, and Sherpa/PNUTS from Yahoo!. Key-value stores that support semi-structured data (also known as document-stores) include Couchbase, DynamoDB from Amazon, and MongoDB, among others. Redis and Memcached are parallel in-memory key-value stores which are widely used for caching data.

Key-value stores are not full-fledged databases, since they do not provide many of the features that are viewed as standard on database systems today. Features that key-value stores typically do not support include declarative querying (using SQL or any other declarative query language), support for transactions, and support for efficient retrieval of records based on selections on nonkey attributes (traditional databases support such retrieval using secondary indices). In fact, they typically do not support primary-key constraints for attributes other than the key, and do not support foreign-key constraints.

21.7.1 Data Representation

As an example of data management needs of web applications, consider the profile of a user, which needs to be accessible to a number of different applications that are run by an organization. The profile contains a variety of attributes, and there are frequent additions to the attributes stored in the profile. Some attributes may contain complex data. A simple relational representation is often not sufficient for such complex data.

Many key-value stores support the *JavaScript Object Notation* (*JSON*) representation, which has found increasing acceptance for representing complex data (JSON is covered in Section 8.1.2). The JSON representation provides flexibility in the set of attributes that a record contains, as well as the types of these attributes. Yet others, such as Bigtable, define their own data model for complex data, including support for records with a very large number of optional columns.

In Bigtable, a record is not stored as a single value but is instead split into component attributes that are stored separately. Thus, the key for an attribute value conceptually consists of (record-identifier, attribute-name). Each attribute value is just a string as far as Bigtable is concerned. To fetch all attributes of a record, a range query, or more precisely a prefix-match query consisting of just the record identifier, is used. The `get()` function returns the attribute names along with the values. For efficient retrieval

Telegram: @uni_k

www.konkur.in

of all attributes of a record, the storage system stores entries sorted by the key, so all attribute values of a particular record are clustered together.

In fact, the record identifier can itself be structured hierarchically, although to Bigtable itself the record identifier is just a string. For example, an application that stores pages retrieved from a web crawl could map a URL of the form:

www.cs.yale.edu/people/silberschatz.html

to the record identifier:

edu.yale.cs.www/people/silberschatz.html

so that pages are clustered in a useful order.

Data-storage systems often allow multiple versions of data items to be stored. Versions are often identified by timestamp, but they may be alternatively identified by an integer value that is incremented whenever a new version of a data item is created. Reads can specify the required version of a data item, or they can pick the version with the highest version number. In Bigtable, for example, a key actually consists of three parts: (record-identifier, attribute-name, timestamp).

Some key-value stores support *columnar storage* of rows, with each column of a row stored separately, with the row key and the column value stored for each row. Such a representation allows a scan to efficiently retrieve a specified column of all rows, without having to retrieve other columns from storage. In contrast, if rows are stored in the usual manner, with all column values stored with the row, a sequential scan of the storage would fetch columns that are not required, reducing performance.

Further, some key-value stores support the notion of a **column family**, which groups sets of columns into a column family. For a given row, all the columns in a specific column family are stored together, but columns from other column families are stored separately. If a set of columns are often retrieved together, storing them as a column family may allow more efficient retrieval, as compared to either columnar storage where these are stored and retrieved separately, or a row storage, which could result in retrieving unneeded columns from storage.

21.7.2 Storing and Retrieving Data

In this section, we use the term *tablet* to refer to partitions, as discussed in Section 21.3.3. We also use the term **tablet server** to refer to the node that acts as the server for a particular tablet; all requests related to a tablet are sent to the tablet server for that tablet.[5]. The tablet server would be one of the nodes that has a replica of the tablet and plays the role of master replica as discussed in Section 21.4.1.[6]

[5] HBase uses the terms *region* and *region server* in place of the terms *tablet* and *tablet server*

[6] In BigTable and HBase, replication is handled by the underlying distributed file system; tablet data are stored in files, and one of the nodes containing a replica of the tablet files is

Telegram: @uni_k

www.konkur.in

We use the term **master** to refer to a site that stores a master copy of the partition information, including, for each tablet, the key ranges for the tablet, the sites storing the replicas of the tablet, and the current tablet server for that tablet.[7] The master is also responsible for tracking the health of tablet servers; in case a tablet server node fails, the master assigns one of the other nodes that contains a replica of the tablet to act as the new tablet server for that tablet. The master is also responsible for reassigning tablets to balance the load in the system if some node is overloaded or if a new node is added to the system.

For each request coming into the system, the tablet corresponding to the key must be identified, and the request routed to the tablet server. If a single master site were responsible for this task, it would get overloaded. Instead, the routing task is parallelized in one of two ways:

- By replicating the partition information to the client sites; the key-value store API used by clients looks up the partition information copy stored at the client to decide where to route a request. This approach is used in Bigtable and HBase.
- By replicating the partition information to a set of router sites, which route requests to the site with the appropriate tablet. Requests can be sent to any one of the router sites, which forward the request to the correct tablet master. This approach is used, for example, in the PNUTS system.

Since there may be a gap between actually splitting or moving a tablet and updating the partition information at a router (or client), the partition information may be out of date when the routing decision is made. When the request reaches the identified tablet master node, the node detects that the tablet has been split, or that the site no longer stores a (master) replica of the tablet. In such a case, the request is returned to the router with an indication that the routing was incorrect; the router then retrieves up-to-date tablet mapping information from the master and reroutes the request to the correct destination.

Figure 21.8 depicts the architecture of a cloud data-storage system, based loosely on the PNUTS architecture. Other systems provide similar functionality, although their architecture may vary. For example, Bigtable/HBase do not have separate routers; the partitioning and tablet-server mapping information is stored in the Google File System/HDFS, and clients read the information from the file system and decide where to send their requests.

21.7.2.1 Geographically Distributed Storage

Several key-value stores support replication of data to geographically distributed locations; some of these also support partitioning of data across geographically distributed locations, allowing different partitions to be replicated in different sets of locations.

[7] The term *tablet controller* is used by PNUTS to refer to the master site.

Telegram: @uni_k

www.konkur.in

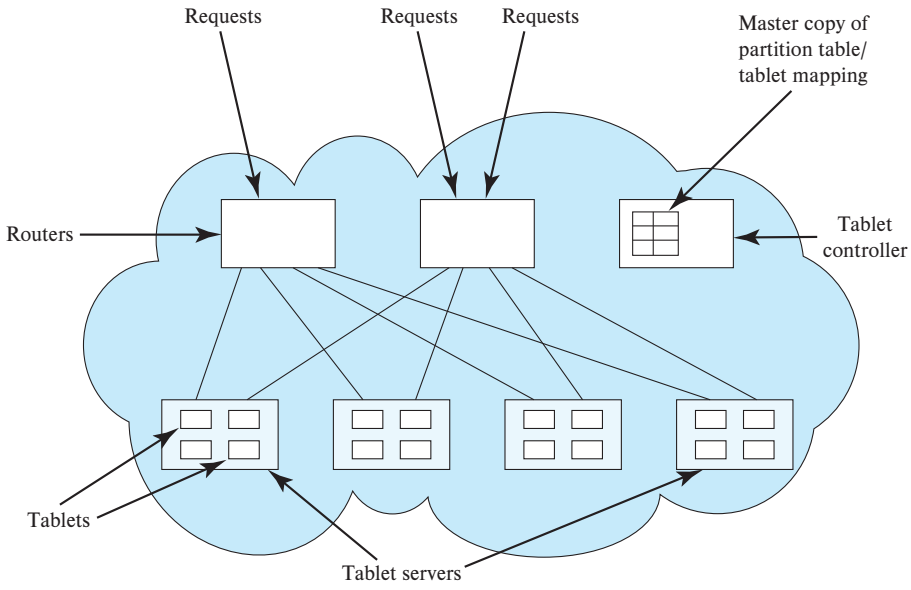


Figure 21.8 Architecture of a cloud data storage system.

One of the key motivations for geographic distribution is fault tolerance, which allows the system to continue functioning even if an entire data center fails due to a disaster such as a fire or an earthquake; in fact, earthquakes could cause all data centers in a region to fail. A second key motivation is to allow a copy of the data to reside at a geographic region close to the user; requiring data to be fetched from across the world could result in latencies of hundreds of milliseconds.

A key performance issue with geographical replication of data is that the latency across geographical regions is much higher than the latency within a data center. Some key-value stores nevertheless support geographically distributed replication, requiring transactions to wait for confirmation of updates from remote locations. Other key-value stores support asynchronous replication of updates to remote locations, allowing a transaction to commit without waiting for confirmation of updates from a remote location. There is, however, a risk of loss of updates in case of failure before the updates are replicated. Some key-value stores allow the application to choose whether to wait for confirmation from remote locations or to commit as soon as updates are performed locally.

Key-value stores that support geographic replication include Apache Cassandra, Megastore and Spanner from Google, Windows Azure storage from Microsoft, and PNUTS/Sherpa from Yahoo!, among others.

Telegram: @uni_k

www.konkur.in

21.7.2.2 Index Structure

The records in each tablet in a key-value store are indexed on the key; range queries can be efficiently supported by storing records clustered on the key. A B^+-tree file organization is a good option, since it supports indexing with clustered storage of records.

The widely used key-value stores BigTable and HBase are built on top of distributed file systems in which files are *immutable*; that is, files cannot be updated once they are created. Thus B^+-tree indices or file organization cannot be stored in immutable files, since B^+-trees require updates, which cannot be done on an immutable file.

Instead, the BigTable and HBase systems use the *stepped-merge* variant of the *log structured merge tree* (*LSM tree*), which we saw in Section 14.8.1, and is described in more detail in Section 24.2. The LSM tree does not perform updates on existing trees, but instead creates new trees either using new data or by merging existing trees. Thus, it is an ideal fit for use on top of distributed file systems that only support immutable files. As an extra benefit, the LSM tree supports clustered storage of records, and can support very high insert and update rates, which has been found very useful in many applications of key-value stores. Several key-value stores, such as Apache Cassandra and the WiredTiger storage structure used by MongoDB, use the LSM tree structure.

21.7.3 Support for Transactions

Most key-value stores offer limited support for transactions. For example, key-value stores typically support atomic updates on a single data item and ensure that updates on the data item are serialized, that is, run one after the other. Serializability at the level of individual operations is thus trivially satisfied, since the operations are run serially. Note that serializability at the level of transactions is not guaranteed by serial execution of updates on individual data items, since a transaction may access more than one data item.

Some key-value stores, such as Google's MegaStore and Spanner, provide full support for ACID transactions across multiple nodes. However, most key-value stores do not support transactions across multiple data items.

Some key-value stores provide a test-and-set operation that can help applications implement limited forms of concurrency control, as we see next.

21.7.3.1 Concurrency Control

Some key-value stores, such as the Megastore and Spanner systems from Google, support concurrency control via locking. Issues in distributed concurrency control are discussed in Chapter 23. Spanner also supports versioning and database snapshots based on timestamps. Details of the multiversion concurrency control technique implemented in Spanner are discussed in Section 23.5.1.

However, most of the other key-value stores, such as Bigtable, PNUTS/Sherpa, and MongoDB, support atomic operations on single data items (which may have multiple columns, or may be JSON documents in MongoDB).

Telegram: @uni_k

www.konkur.in

Some key-value stores, such as HBase and PNUTS, provide an atomic test-and-set function, which allows an update to a data item to be conditional on the current version of the data item being the same as a specified version number; the check (test) and the update (set) are performed atomically. This feature can be used to implement a limited form of validation-based concurrency control, as discussed in Section 23.3.7.

Some data stores support atomic increment operations on data items and atomic execution of stored procedures. For example, HBase supports the incrementColumnValue() operation, which atomically reads and increments a column value, and a checkAndPut() which atomically checks a condition on a data item and updates it only if the check succeeds. HBase also supports atomic execution of stored procedures, which are called "coprocessors" in HBase terminology. These procedures run on a single data item and are executed atomically.

21.7.3.2 Atomic Commit

BigTable, HBase, and PNUTS support atomic commit of multiple updates to a single row; however, none of these systems supports atomic updates across different rows.

As one of the results of the above limitation, none of these systems supports secondary indices; updates to a data item would require updates to the secondary index, which cannot be done atomically.

Some systems, such as PNUTS, support secondary indices or materialized views with deferred updates; updates to a data item result in updates to the secondary index or materialized view being added to a messaging service to be delivered to the node where the update needs to be applied. These updates are guaranteed to be delivered and applied subsequently; however, until they are applied, the secondary index may be inconsistent with the underlying data. View maintenance is also supported by PNUTS in the same deferred fashion. There is no transactional guarantee on the updates of such secondary indices or materialized views, and only a best-effort guarantee in terms of when the updates reach their destination. Consistency issues with deferred maintenance are discussed in Section 23.6.3.

In contrast, the Megastore and Spanner systems developed by Google support atomic commit for transactions spanning multiple data items, which can be spread across multiple nodes. These systems use two-phase commit (discussed in Section 23.2) to ensure atomic commit across multiple nodes.

21.7.3.3 Dealing with Failures

If a tablet server node fails, another node that has a copy of the tablet should be assigned the task of serving the tablet. The master node is responsible for detecting node failures and reassigning tablet servers.

When a new node takes over as tablet server, it must recover the state of the tablet. To ensure that updates to the tablet survive node failures, updates to a tablet are logged, and the log is itself replicated. When a site fails, the tablets at the site are assigned to

Telegram: @uni_k

www.konkur.in

other sites; the new master site of each tablet is responsible for performing recovery actions using the log to bring its copy of the tablet to an up-to-date state, after which updates and reads can be performed on the tablet.

In Bigtable, as an example, mapping information is stored in an index structure, and the index, as well as the actual tablet data, are stored in the file system. Tablet data updates are not flushed immediately, but log data are. The file system ensures that the file system data are replicated and will be available even in the face of failure of a few nodes in the cluster. Thus, when a tablet is reassigned, the new server for that tablet has access to up-to-date log data.

Yahoo!'s Sherpa/PNUTS system, on the other hand, explicitly replicates tablets to multiple nodes in a cluster and uses a persistent messaging system to implement the log. The persistent messaging system replicates log records at multiple sites to ensure availability in the event of a failure. When a new node takes over as the tablet server, it must apply any pending log records that were generated by the earlier tablet server before taking over as the tablet server.

To ensure availability in the face of failures, data must be replicated. As noted in Section 21.4.2, a key issue with replication is the task of keeping the replicas consistent with each other. Different systems implement atomic update of replicas in different fashions. Google BigTable and Apache HBase use replication features provided by an underlying file system (GFS for BigTable, and HDFS for HBase), instead of implementing replication on their own. Interestingly, neither GFS nor HDFS supports atomic updates of all replicas of a file; instead they support appends to files, which are copied to all replicas of the file blocks. An append is successful only when it has been applied to all replicas. System failures can result in appends that are applied to only some replicas; such incomplete appends are detected using sequence numbers and are cleaned up when they are detected.

Some systems such as PNUTS use a persistent messaging service to log updates; the messaging service guarantees that updates will be delivered to all replicas. Other systems, such as Google's Megastore and Spanner, use a technique called distributed consensus to implement consistent replication, as we discuss in Section 23.8. Such systems require a majority of replicas to be available to perform an update. Other systems, such as Apache Cassandra and MongoDB, allow the user control over how many replicas must be available to perform an update. Setting the value low could result in conflicting updates, which must be resolved later. We discuss these issues in Section 23.6.

21.7.4 Managing Without Declarative Queries

Key-value stores do not provide any query processing facility, such as SQL language support, or even lower-level primitives such as joins. Many applications that use key-value stores can manage without query language support. The primary mode of data access in such applications is to store data with an associated key and to retrieve data

Telegram: @uni_k

www.konkur.in

with that key. In the user profile example, the key for user-profile data would be the user's identifier.

There are applications that require joins but implement the joins either in application code or by a form of view materialization. For example, in a social-networking application, each user should be shown new posts from all her friends, which conceptually requires a join.

One approach to computing the join is to implement it in the application code, by first finding the set of friends of a given user, and then querying the data object representing each friend, to find their recent posts.

An alternative approach is as follows: Whenever a user makes a post, for each friend of the user a message is sent to, the data object representing that friend and the data associated with the friend are updated with a summary of the new post. When that user checks for updates, all required data are available in one place and can be retrieved quickly.

Both approaches can be used without any underlying support for joins. There are trade-offs between the two alternatives such as higher cost at query time for the first alternative versus higher storage cost and higher cost at the time of writes for the second alternative.

21.7.5 Performance Optimizations

When using a data storage system, the physical location of data are decided by the storage system and hidden from the client. When storing multiple relations that need to be joined, partitioning each independently may be suboptimal in terms of communication cost. For example, if the join of two relations is computed frequently, it may be best if they are partitioned in exactly the same way, on their join attributes. As we will see in Section 22.7.4, doing so would allow the join to be computed in parallel at each storage site, without data transfer.

To support such scenarios, some data storage systems allow the schema designer to specify that tuples of one relation should be stored in the same partitions as tuples of another relation that they reference, typically using a foreign key. A typical use of this functionality is to store all tuples related to a particular entity together in the same partition; the set of such tuples is called an **entity group**.

Further, many data storage systems, such as HBase, support *stored functions* or *stored procedures*. Stored functions/procedures allow clients to invoke a function on a tuple (or an entity group) and instead of the tuples being fetched and executed locally, the function is executed at the partition where the tuple is stored. Stored functions/procedures are particularly useful if the stored tuples are large, while the function/procedure results are small, reducing data transfer.

Many data storage systems provide features such as support for automatically deleting old versions of data items after some period of time, or even deleting data items that are older than some specified period.

Telegram: @uni_k

www.konkur.in

21.8 Summary

- Parallel databases have gained significant commercial acceptance in the past 20 years.
- Data storage and indexing are two important aspects of parallel database systems.
- Data partitioning involves the distribution of data among multiple nodes. In I/O parallelism, relations are partitioned among available disks so that they can be retrieved faster. Three commonly used partitioning techniques are round-robin partitioning, hash partitioning, and range partitioning.
- Skew is a major problem, especially with increasing degrees of parallelism. Balanced partitioning vectors, using histograms, and virtual node partitioning are among the techniques used to reduce skew.
- Parallel data storage systems must be resilient to failure of nodes. To ensure that data are not lost on node failure, tuples are replicated across at least two nodes, and often three nodes. If a node fails, the tuples that it stored can still be accessed from the other nodes where the tuples are replicated.
- Indices in a parallel data storage system can be divided into two kinds: local and global. A local index is an index built on tuples stored in a particular node; The index contents are stored on the same node as the data. A global index is an index built on data stored across multiple nodes.
- A distributed file system stores files across a large collection of machines, while giving a single-file-system view to clients. As with any file system, there is a system of file names and directories, which clients can use to identify and access files. Clients do not need to bother about where the files are stored.
- Web applications need to store very large numbers (many billions) of relatively small records (of size ranging from a few kilobytes to a few megabytes). A number of parallel key-value storage systems were developed to meet the needs of such web applications. A key-value store provides a way to store or update a data item (value) with an associated key, and to retrieve the data item with a given key.

Review Terms

- Key-value stores
- Data storage system
- I/O parallelism
- Data partitioning
- Horizontal partitioning
- Partitioning strategies
  - Round-robin
  - Hash partitioning
  - Range partitioning
- Partitioning vector

Telegram: @uni_k

www.konkur.in

- Point queries
- Range queries
- Skew
  - Execution skew
  - Data distribution skew
  - Attribute-value skew
  - Partition skew
  - Execution skew
- Handling of skew
  - Balanced range-partitioning vector
  - Histogram
  - Virtual nodes
- Elasticity of storage
- Table
- Tablets
- Partition table
- Master node
- Routers
- Consistent hashing
- Distributed hash tables
- Replication
  - Replication within a data center
  - Replication across data center
  - Master replicas
  - Consistency of replicas
- Eventual consistency
- Global primary index
- Global secondary index
- Distributed file system
- Write-once-read-many access model
- Key-value store
- Wide-column stores
- Document stores
- Column family
- Tablet server

Practice Exercises

21.1 In a range selection on a range-partitioned attribute, it is possible that only one disk may need to be accessed. Describe the benefits and drawbacks of this property.

21.2 Recall that histograms are used for constructing load-balanced range partitions.

a. Suppose you have a histogram where values are between 1 and 100, and are partitioned into 10 ranges, 1 – 10, 11 – 20, … , 91 – 100, with frequencies 15, 5, 20, 10, 10, 5, 5, 20, 5, and 5, respectively. Give a load-balanced range partitioning function to divide the values into five partitions.

b. Write an algorithm for computing a balanced range partition with p partitions, given a histogram of frequency distributions containing n ranges.

21.3 Histograms are traditionally constructed on the values of a specific attribute (or set of attributes) of a relation. Such histograms are good for avoiding data

Telegram: @uni_k

www.konkur.in

distribution skew but are not very useful for avoiding execution skew. Explain why.

Now suppose you have a workload of queries that perform point lookups. Explain how you can use the queries in the workload to come up with a partitioning scheme that avoids execution skew.

21.4 Replication:

a. Give two reasons for replicating data across geographically distributed data centers.

b. Centralized databases support replication using log records. How is the replication in centralized databases different from that in parallel/distributed databases?

21.5 Parallel indices:

a. Secondary indices in a centralized database store the record identifier. A global secondary index too could potentially store a partition number holding the record, and a record identifier within the partition. Why would this be a bad idea?

b. Global secondary indices are implemented in a way similar to local secondary indices that are used when records are stored in a B^+-tree file organization. Explain the similarities between the two scenarios that result in a similar implementation of the secondary indices.

21.6 Parallel database systems store replicas of each data item (or partition) on more than one node.

a. Why is it a good idea to distribute the copies of the data items allocated to a node across multiple other nodes, instead of storing all the copies in the same node (or set of nodes).

b. What are the benefits and drawbacks of using RAID storage instead of storing an extra copy of each data item?

21.7 Partitioning and replication.

a. Explain why range-partitioning gives better control on tablet sizes than hash partitioning. List an analogy between this case and the case of B^+-tree indices versus hash indices.

b. Some systems first perform hashing on the key, and then use range partitioning on the hash values. What could be a motivation for this choice, and what are its drawbacks as compared to performing range partition direction on the key?

c. It is possible to horizontally partition data, and then perform vertical partitioning locally at each node. It is also possible to do the converse,

Telegram: @uni_k

www.konkur.in

where vertical partitioning is done first, and then each partition is then horizontally partitioned independently. What are are the benefits of the first option over the second one?

21.8 In order to send a request to the master replica of a data item, a node must know which replica is the master for that data item.

a. Suppose that between the time the node identifies which node is the master replica for a data item, and the time the request reaches the identified node, the mastership has changed, and a different node is now the master. How can such a situation be dealt with?

b. While the master replica could be chosen on a per-partition basis, some systems support a *per-record master replica*, where the records of a partition (or tablet) are replicated at some set of nodes, but each record's master replica can be on any of the nodes from within this set of nodes, independent of the master replica of other records. List two benefits of keeping track of master on a per-record basis.

c. Suggest how to keep track of the master replica for each record, when there are a large number of records.

Exercises

21.9 For each of the three partitioning techniques, namely, round-robin, hash partitioning, and range partitioning, give an example of a query for which that partitioning technique would provide the fastest response.

21.10 What factors could result in skew when a relation is partitioned on one of its attributes by:

a. Hash partitioning?

b. Range partitioning?

In each case, what can be done to reduce the skew?

21.11 What is the motivation for storing related records together in a key-value store? Explain the idea using the notion of an entity group.

21.12 Why is it easier for a distributed file system such as GFS or HDFS to support replication than it is for a key-value store?

21.13 Joins can be expensive in a key-value store, and difficult to express if the system does not support SQL or a similar declarative query language. What can an application developer do to efficiently get results of join or aggregate queries in such a setting?

Telegram: @uni_k

www.konkur.in

Tools

A wide variety of open-source Big Data tools are available, in addition to some commercial tools. In addition, a number of these tools are available on cloud platforms. Google File System (GFS) was an early generation parallel file system. Apache HDFS (hadoop.apache.org) is a widely used distributed file system implementation modeled after GFS. HDFS by itself does not define any internal format for files, but Hadoop implementations today support several optimized file formats such as Sequence files (which allow binary data), Avro (which supports semi-structured schemas) and Parquet and Orc (which support columnar data representation). Hosted cloud storage systems include the Amazon S3 storage system (aws.amazon.com/s3) and Google Cloud Storage (cloud.google.com/storage).

Google's Bigtable was an early generation parallel data storage system, architected as a layer on top of GFS. Amazon's Dynamo is an early generation parallel key-value store which is based on the idea of consistent hashing, developed initially for peer-to-peer data storage. Both are available hosted on the cloud as Google Bigtable (cloud.google.com/bigtable) and Amazon DynamoDB (aws.amazon.com/dynamodb). Google Spanner (cloud.google.com/spanner) is a hosted storage system that provides extensive transactional support. Apache HBase (hbase.apache.org) is a widely used open-source data storage system which is based on Bigtable and is implemented as a layer on top of HDFS. Apache Cassandra (cassandra.apache.org) which was developed at Facebook, Voldemort (www.project-voldemort.com) developed at LinkedIn, MongoDB (www.mongodb.com), CouchDB (couchdb.apache.org) and Riak (basho.com) are all open-source key-value stores. MongoDB and CouchDB use the JSON format for storing data. Aerospike (www.aerospike.com) is an open-source data storage system optimized for Flash storage. There are many other open-source parallel data storage systems available today.

Commercial parallel database systems include Teradata, Teradata Aster Data, IBM Netezza, and Pivotal Greenplum. IBM Netezza, Pivotal Greenplum, and Teradata Aster Data all use PostgreSQL as the underlying database, running independently on each node; each of these systems builds a layer on top, to partition data, and parallelize query processing across the nodes.

Further Reading

In the late 1970s and early 1980s, as the relational model gained reasonably sound footing, people recognized that relational operators are highly parallelizable and have good dataflow properties. Several research projects, including GAMMA ([DeWitt (1990)]), XPRS ([Stonebraker et al. (1988)]), and Volcano ([Graefe (1990)]) were launched to investigate the practicality of parallel storage of data and parallel execution of queries.

Teradata was one of the first commercial shared-nothing parallel database systems designed for decision support systems, and it continues to have a large market share.

Telegram: @uni_k

www.konkur.in

Teradata supports partitioning and replication of data to deal with node failures. The Red Brick Warehouse was another early parallel database system designed for decision support (Red Brick was bought by Informix, and later IBM).

Information on the Google file system can be found in [Ghemawat et al. (2003)], while the Google Bigtable system is described in [Chang et al. (2008)]. The Yahoo! PNUTS system is described in [Cooper et al. (2008)], while Google Megastore and Google Spanner are described in [Baker et al. (2011)] and [Corbett et al. (2013)] respectively. Consistent hashing is described in [Karger et al. (1997)], while Dynamo, which is based on consistent hashing, is described in [DeCandia et al. (2007)].

Bibliography

[Baker et al. (2011)] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services", In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pages 223–234.

[Chang et al. (2008)] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", *ACM Trans. Comput. Syst.*, Volume 26, Number 2 (2008).

[Cooper et al. (2008)] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform", *Proceedings of the VLDB Endowment*, Volume 1, Number 2 (2008), pages 1277–1288.

[Corbett et al. (2013)] J. C. Corbett et al., "Spanner: Google's Globally Distributed Database", *ACM Trans. on Computer Systems*, Volume 31, Number 3 (2013).

[DeCandia et al. (2007)] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazons Highly Available Key-value Store", In *Proc. of the ACM Symposium on Operating System Principles* (2007), pages 205–220.

[DeWitt (1990)] D. DeWitt, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Number 1 (1990), pages 44–62.

[Ghemawat et al. (2003)] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *Proc. of the ACM Symposium on Operating System Principles* (2003).

[Graefe (1990)] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 102–111.

[Karger et al. (1997)] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", In *Proc. of the ACM Symposium on Theory of Computing* (1997), pages 654–663.

Telegram: @uni_k

www.konkur.in

[Stonebraker et al. (1988)] M. Stonebraker, R. H. Katz, D. A. Patterson, and J. K. Ousterhout, "The Design of XPRS", In *Proc. of the International Conf. on Very Large Databases* (1988), pages 318–330.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 22



Parallel and Distributed Query Processing

In this chapter, we discuss algorithms for query processing in parallel database systems. We assume that the queries are **read only**, and our focus is on query processing in decision support systems. Such systems need to execute queries on very large amounts of data, and parallel processing of the query across multiple nodes is critical for processing queries within acceptable response times.

Our focus in the early parts of this chapter is on relational query processing. However, later in the chapter, we examine issues in parallel processing of queries expressed in models other than the relational model.

Transaction processing systems execute large numbers of queries that perform updates, but each query affects only a small number of tuples. Parallel execution is key to handle large transaction processing loads; however, this topic is covered in Chapter 23.

22.1 Overview

Parallel processing can be exploited in two distinct ways in a database system. One approach is **interquery parallelism**, which refers to the execution of multiple queries in parallel with each other, across multiple nodes. The second approach is **intraquery parallelism**, which refers to the processing of different parts of the execution of a single query, in parallel across multiple nodes.

Interquery parallelism is essential for transaction processing systems. Transaction throughput can be increased by this form of parallelism. However, the response times of individual transactions are no faster than they would be if the transactions were run in isolation. Thus, the primary use of interquery parallelism is to scale up a transaction-processing system to support a larger number of transactions per second. Transaction processing systems are considered in Chapter 23.

In contrast, intraquery parallelism is essential for speeding up long-running queries, and it is the focus of this chapter.

Telegram: @uni_k

www.konkur.in

Execution of a single query involves execution of multiple operations, such as selects, joins, or aggregate operations. The key to exploiting large-scale parallelism is to process each operation in parallel, across multiple nodes. Such parallelism is referred to as **intraoperation parallelism**. Since the number of tuples in a relation can be large, the degree of intraoperation parallelism is also potentially very large; thus, intraoperation parallelism is natural in a database system.

To illustrate the parallel evaluation of a query, consider a query that requires a relation to be sorted. Suppose that the relation has been partitioned across multiple disks by range partitioning on some attribute, and the sort is requested on the partitioning attribute. The sort operation can be implemented by sorting each partition in parallel, then concatenating the sorted partitions to get the final sorted relation. Thus, we can parallelize a query by parallelizing individual operations.

There is another source of parallelism in evaluating a query: The *operator tree* for a query can contain multiple operations. We can parallelize the evaluation of the operator tree by evaluating in parallel some of the operations that do not depend on one another. Further, as Chapter 15 mentions, we may be able to pipeline the output of one operation to another operation. The two operations can be executed in parallel on separate nodes, one generating output that is consumed by the other, even as it is generated. Both these forms of parallelism are examples of **interoperation parallelism**, which allows different operators of a query to be executed in parallel.

In summary, the execution of a single query can be parallelized in two different ways:

- **Intraoperation parallelism**, which we consider in detail in the next few sections, where we study parallel implementations of common relational operations such as sort, join, aggregate and other operations.
- **Interoperation parallelism**, which we consider in detail in Section 22.5.1.

The two forms of parallelism are complementary and can be used simultaneously on a query. Since the number of operations in a typical query is small, compared to the number of tuples processed by each operation, intraoperation parallelism can scale better with increasing parallelism. However, interoperation parallelism is also important, especially in shared memory systems with multiple cores.

To simplify the presentation of the algorithms, we assume a shared nothing architecture with n nodes, $N_1, N_2, \ldots, N_n$. Each node may have one or more disks, but typically the number of such disks is small. We do not address how to partition the data between the disks at a node; RAID organizations can be used with these disks to exploit parallelism at the storage level, rather than at the query processing level.

The choice of algorithms for parallelizing query evaluation depends on the machine architecture. Rather than present algorithms for each architecture separately, we use a shared-nothing architecture in our description. Thus, we explicitly describe when data have to be transferred from one node to another.

Telegram: @uni_k

www.konkur.in

We can simulate this model easily by using the other architectures, since transfer of data can be done via shared memory in a shared-memory architecture, and via shared disks in a shared-disk architecture. Hence, algorithms for shared-nothing architectures can be used on the other architectures too. In Section 22.6, we discuss how some of the algorithms can be further optimized for shared-memory systems.

Current-generation parallel systems are typically based on a hybrid architecture, where each computer has multiple cores with a shared memory, and there are multiple computers organized in a shared-nothing fashion. For the purpose of our discussion, with such an architecture, each core can be considered a node in a shared-nothing system. Optimizations to exploit the fact that some of the cores share memory with other cores can be performed as discussed in Section 22.6.

22.2 Parallel Sort

Suppose that we wish to sort a relation r that resides on n nodes $N_1, N_2, \ldots, N_n$. If the relation has been range-partitioned on the attributes on which it is to be sorted, we can sort each partition separately and concatenate the results to get the full sorted relation. Since the tuples are partitioned on n nodes, the time required for reading the entire relation is reduced by a factor of n by the parallel access.

If relation r has been partitioned in any other way, we can sort it in one of two ways:

1. We can range-partition r on the sort attributes, and then sort each partition separately.
2. We can use a parallel version of the external sort-merge algorithm.

22.2.1 Range-Partitioning Sort

Range-partitioning sort, shown pictorially in Figure 22.1a, works in two steps: first range-partitioning the relation, then sorting each partition separately. When we sort by range-partitioning the relation, it is not necessary to range-partition the relation on the same set of nodes as those on which that relation is stored. Suppose that we choose nodes $N_1, N_2, \ldots, N_m$ to sort the relation. There are two steps involved in this operation:

1. Redistribute the tuples in the relation, using a range-partition strategy, so that all tuples that lie within the ith range are sent to node n_i, which stores the relation temporarily on its local disk.

   To implement range partitioning, in parallel every node reads the tuples from its disk and sends each tuple to its destination node based on the partition function. Each node $N_1, N_2, \ldots, N_m$ also receives tuples belonging to its partition and stores them locally. This step requires disk and network communication.

Telegram: @uni_k

www.konkur.in

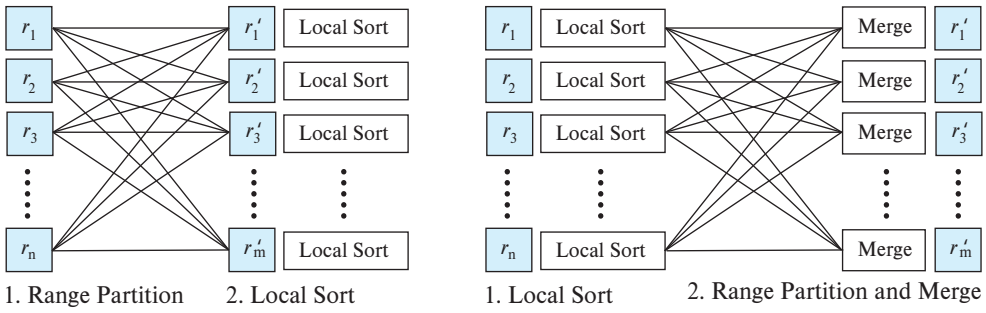


Figure 22.1 Parallel sorting algorithms.

2. Each of the nodes sorts its partition of the relation locally, without interaction with the other nodes. Each node executes the same operation—namely, sorting—on a different data set. (Execution of the same operation in parallel on different sets of data are called **data parallelism**.)

   The final merge operation is trivial, because the range partitioning in the first phase ensures that, for $1 \leq i < j \leq m$, the key values in node N_i are all less than the key values in N_j.

We must do range partitioning with a balanced range-partition vector so that each partition will have approximately the same number of tuples. We saw how to create such partition vectors in Section 21.3.1. Virtual node partitioning, as discussed in Section 21.3.2, can also be used to reduce skew. Recall that there are several times as many virtual nodes as real nodes, and virtual node partitioning creates a partition for each virtual node. Virtual nodes are then mapped to real nodes; doing so in a round-robin fashion tends to spreads virtual nodes across real nodes in a way that reduces the degree of skew at real nodes.

22.2.2 Parallel External Sort-Merge

Parallel external sort-merge, shown pictorially in Figure 22.1b, is an alternative to range partitioning sort. Suppose that a relation has already been partitioned among nodes $N_1, N_2, \ldots, N_n$ (it does not matter how the relation has been partitioned). Parallel external sort-merge then works this way:

1. Each node N_i sorts the data available at N_i.
2. The system then merges the sorted runs on each node to get the final sorted output.

Telegram: @uni_k

www.konkur.in

The merging of the sorted runs in step 2 can be parallelized by this sequence of actions:

1. The system range-partitions the sorted partitions at each node N_i (all by the same partition vector) across the nodes $N_1, N_2, \ldots, N_m$. It sends the tuples in sorted order, so each node receives the tuples as sorted streams.
2. Each node N_i performs a merge on the streams of tuples as they are received to get a single sorted run.
3. The system concatenates the sorted runs on nodes $N_1, N_2, \ldots, N_m$ to get the final result.

As described, this sequence of actions results in an interesting form of **execution skew**, since at first every node sends all tuples of partition 1 to N_1, then every node sends all tuples of partition 2 to N_2, and so on. Thus, while sending happens in parallel, receiving tuples becomes sequential: First only N_1 receives tuples, then only N_2 receives tuples, and so on. To avoid this problem, the sorted sequence of tuples $S_{i,j}$ from any node i destined to any other node j is broken up into multiple blocks. Each node N_i sends the first block of tuples from $S_{i,j}$ node N_j, for each j; it then sends the second block of tuples to each node N_j, and so on, until all blocks have been sent. As a result, all nodes receive data in parallel. (Note that tuples are sent in blocks, rather than individually, to reduce network overheads.)

22.3 Parallel Join

Parallel join algorithms attempt to divide the tuples of the input relations over several nodes. Each node then computes part of the join locally. Then, the system collects the results from each node to produce the final result. How exactly to divide the tuples depends on the join algorithm, as we see next.

22.3.1 Partitioned Join

For certain kinds of joins, it is possible to *partition* the two input relations across the nodes and to compute the join locally at each node. The partitioned join technique can be used for inner joins, where the join condition is an equi-join (e.g., $r \bowtie_{r.A=s.B} s$); the relations r and s are partitioned by the same partitioning function on their join attributes. The idea of partitioning is exactly the same as that behind the partitioning step of hash join. Partitioned join can also be used for outer joins, as we shall see shortly.

Suppose that we are using m nodes to perform the join, and that the relations to be joined are r and s. **Partitioned join** then works this way: The system partitions the relations r and s each into m partitions, denoted $r_1, r_2, \ldots, r_m$ and $s_1, s_2, \ldots, s_m$. In a partitioned join, however, there are two different ways of partitioning r and s:

Telegram: @uni_k

www.konkur.in

- Range partitioning on the join attributes.
- Hash partitioning on the join attributes.

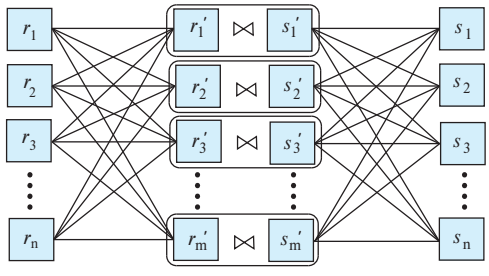
In either case, the same partitioning function must be used for both relations. For range partitioning, the same partition vector must be used for both relations. For hash partitioning, the same hash function must be used on both relations. Figure 22.2 depicts the partitioning in a partitioned parallel join.

The partitioned join algorithm first partitions one of the relations by scanning its tuples and sending them to the appropriate node based on the partition function and the join attribute values of each tuple. Specifically, each node N_i reads in the tuples of one of the relations, say r, from local disk, computes for each tuple t the partition r_j to which t belongs, and sends the tuple t to node N_j. Each node also simultaneously receives tuples that are sent to it and stores them on its local disk (this can be done by having separate threads for sending and receiving data). The process is repeated for all tuples from the other relation, s.

Once both relations are partitioned, we can use any join technique locally at each node N_i to compute the join of r_i and s_i. Thus, we can use partitioning to parallelize any join technique.

Partitioned join can be used not only for inner joins, but also for all three forms of outer join (left, right and full outer join). Each node computes the corresponding outer join locally, after partitioning is done on the join attributes. Further, since natural join can be expressed as an equijoin followed by a projection, natural joins can also be computed using partitioned join.

If one or both of the relations r and s are already partitioned on the join attributes (by either hash partitioning or range partitioning), the work needed for partitioning is reduced greatly. If the relations are not partitioned or are partitioned on attributes other than the join attributes, then the tuples need to be repartitioned.



Step 1: Partition r Step 2: Partition s

Step 3: Each node N_i computes $r_i' \bowtie s_i'$

Figure 22.2 Partitioned parallel join.

Telegram: @uni_k

www.konkur.in

We now consider issues specific to the join technique used locally at each node N_i. The local join operation can be optimized by performing some initial steps on tuples as they arrive at a node, instead of first storing the tuples to disk and then reading them back to perform these initial steps. These optimizations, which we describe below, are also used in nonparallel query processing, when results of an earlier operation are pipelined into a subsequent operation; thus, they are not specific to parallel query processing.

- If we use hash join locally, the resultant parallel join technique is called **partitioned parallel hash join**.

  Recall that hash join first partitions both input relations into smaller pieces such that each partition of the smaller relation (the build relation) fits into memory. Thus, to implement hash join, the partitions r_i and s_i received by node N_i must be repartitioned using a hash function, say $h1()$. If the partitioning of r and s across the nodes was done by using a hash function $h0()$, the system must ensure that $h1()$ is different from $h0()$. Let the resultant partitions at node N_i be $r_{i,j}$ and $s_{i,j}$ for $j = 1 \ldots n_i$, where n_i denotes the number of local partitions at node N_i.

  Note that the tuples can be repartitioned based on the hash function used for the local hash join as they arrive and written out to the appropriate partitions, avoiding the need to write the tuples to disk and read them back in.

  Recall also that hash join then loads each partition of the build relation into memory, builds an in-memory index on the join attributes, and finally probes the in-memory index using each tuple of the other relation, called the probe relation. Assume that relation s is chosen as the build relation. Then each partition $s_{i,j}$ is loaded in memory, with an index built on the join attributes, and the index is probed with each tuple of $r_{i,j}$.

  Hybrid hash join (described in Section 15.5.5.5) can be used in case the partitions of one of the relations are small enough that a significant part of the partition fits in memory at each node. In this case, the smaller relation, say s, which is used as the build relation, should be partitioned first, followed by the larger relation, say r, which is used as the probe relation. Recall that with hybrid hash join, the tuples in the partition s_0 of the build relation s are retained in memory, and an in-memory index is built on these tuples. When the probe relation tuples arrive at the node, they are also repartitioned; tuples in the r_0 partition are used directly to probe the index on the s_0 tuples, instead of being written out to disk and read back in.

- If we use merge join locally, the resultant technique is called **partitioned parallel merge join**. Each of the partitions s_i and r_i must be sorted, and merged locally, at node N_i.

  The first step of sorting, namely, run generation, can directly consume incoming tuples to generate runs, avoiding a write to disk before run generation.

- If we use nested-loops or indexed nested-loops join locally, the resultant technique is called **partitioned parallel nested-loop join** or **partitioned parallel indexed nested-**

Telegram: @uni_k

www.konkur.in

loops join. Each node N_i performs a nested-loops (or indexed nested-loops) join on s_i and r_i.

22.3.2 Fragment-and-Replicate Join

Partitioning is not applicable to all types of joins. For instance, if the join condition is an inequality, such as $r \bowtie_{r.a<s.b} s$, it is possible that all tuples in r join with some tuple in s (and vice versa). Thus, there may be no nontrivial way of partitioning r and s so that tuples in partition r_i join with only tuples in partition s_i.

We can parallelize such joins by using a technique called *fragment-and-replicate*. We first consider a special case of fragment-and-replicate—**asymmetric fragment-and-replicate join**—which works as follows:

1. The system partitions one of the relations—say, r. Any partitioning technique can be used on r, including round-robin partitioning.
2. The system replicates the other relation, s, across all the nodes.
3. Node N_i then locally computes the join of r_i with all of s, using any join technique.

The asymmetric fragment-and-replicate scheme appears in Figure 22.3a. If r is already stored by partitioning, there is no need to partition it further in step 1. All that is required is to replicate s across all nodes.

The asymmetric fragment-and-replicate join technique is also referred to as **broadcast join**. It is a very useful technique, even for equi-joins, if one of the relations, say s, is small, and the other relation, say r, is large, since replicating the small relation s across all nodes may be cheaper than repartitioning the large relation r.

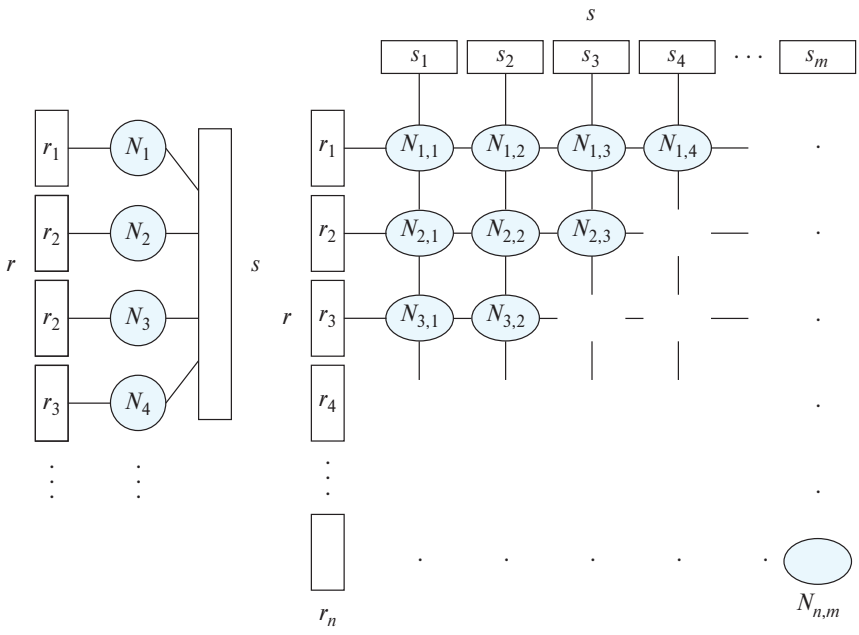
The general case of **fragment-and-replicate join** (also called the **symmetric fragment-and-replicate join** appears in Figure 22.3b; it works this way: The system partitions relation r into n partitions, $r_1, r_2, \ldots, r_n$, and partitions s into m partitions, $s_1, s_2, \ldots, s_m$. As before, any partitioning technique may be used on r and on s. The values of m and n do not need to be equal, but they must be chosen so that there are at least $m * n$ nodes. Asymmetric fragment-and-replicate is simply a special case of general fragment-and-replicate, where $m = 1$. Fragment-and-replicate reduces the sizes of the relations at each node, compared to asymmetric fragment-and-replicate.

Let the nodes be $N_{1,1}, N_{1,2}, \ldots, N_{1,m}, N_{2,1}, \ldots, N_{n,m}$. Node $N_{i,j}$ computes the join of r_i with s_j. To ensure that each node $N_{i,j}$ gets all tuples of r_i and s_j, the system replicates r_i to nodes $N_{i,1}, N_{i,2}, \ldots, N_{i,m}$ (which form a row in Figure 22.3b), and replicates s_i to nodes $N_{1,i}, N_{2,i}, \ldots, N_{n,i}$ (which form a column in Figure 22.3b). Any join technique can be used at each node $N_{i,j}$.

Fragment-and-replicate works with any join condition, since every tuple in r can be tested with every tuple in s. Thus, it can be used where partitioning cannot be used. However, note that each tuple in r is replicated m times, and each tuple in s is replicated n times.

Telegram: @uni_k

www.konkur.in



(a) Asymmetric fragment and replicate

(b) Fragment and replicate

Figure 22.3 Fragment-and-replicate schemes.

Fragment-and-replicate join has a higher cost than partitioning, since it involves replication of both relations, and is therefore used only if the join does not involve equi-join conditions. Asymmetric fragment-and-replicate, on the other hand, is useful even for equi-join conditions, if one of the relations is small, as discussed earlier.

Note that asymmetric fragment-and-replicate join can be used to compute the left outer join operation $r \mathbin{⟕}_\theta s$ if s is replicated, by simply computing the left outer join locally at each node. There is no restriction on the join condition θ.

However, $r \mathbin{⟕}_\theta s$ cannot be computed locally if s is fragmented and r is replicated, since an r tuple may have no matching tuple in partition s_i, but may have a matching tuple in partition $s_j, j \neq i$. Thus, a decision on whether or not to output the r tuple with null values for s attributes cannot be made locally at node N_i. For the same reason, asymmetric fragment-and-replicate cannot be used to compute the full outer join operation, and symmetric fragment-and-replicate cannot be used to compute any of the outer join operations.

22.3.3 Handling Skew in Parallel Joins

Skew presents a special problem for parallel join techniques. If one of the nodes has a much heavier load than other nodes, the parallel join operation will take much longer to finish, with many idle nodes waiting for the heavily loaded node to finish its task.

Telegram: @uni_k

www.konkur.in

When partitioning data for storage, to minimize skew in storage we use a balanced partitioning vector that ensures all nodes get the same number of tuples. For parallel joins, we need to instead balance the execution time of join operations across all nodes. Hash partitioning using any good hash function usually works quite well at balancing the load across nodes, unless some join attribute values occur very frequently. Range partitioning, on the other hand, is more vulnerable to join skew, unless the ranges are carefully chosen to balance the load.[1]

Virtual-node partitioning with, say, round-robin distribution of virtual nodes to real nodes, can help in reducing skew at the level of real nodes even if there is skew at the level of virtual nodes, since the skewed virtual nodes tend to get spread over multiple real nodes.

The preceding techniques are examples of **join skew avoidance**. Virtual-node partitioning, in particular, is very effective at skew avoidance in most cases.

However, there are cases with high skew, for example where some join attribute values are very frequent in both input relations, leading to a large join result size. In such cases, there could be significant join skew, even with virtual-node partitioning.

Dynamic handling of join skew is an alternative to skew avoidance. A dynamic approach can be used to detect and handle skew in such situations. Virtual node partitioning is used, and the system then monitors the join progress at each real node. Each real node schedules one virtual node at a time. Suppose that some real node has completed join processing for all virtual nodes assigned to it, and is thus idle, while some other real node has multiple virtual nodes waiting to be processed. Then, the idle node can get a copy of the data corresponding to one of the virtual nodes at the busy node and process the join for that virtual node. This process can be repeated whenever there is an idle real node, as long as some real node has virtual nodes waiting to be processed.

This technique is an example of **work stealing**, where a processor that is idle takes work that is in the queue of another processor that is busy. Work stealing is inexpensive in a shared-memory system, since all data can be accessed quickly from the shared memory, as discussed further in Section 22.6. In a shared-nothing environment, data movement may be required to move a task from one processor to another, but it is often worth paying the overhead to reduce the completion time of a task.

22.4 Other Operations

In this section, we discuss parallel processing of other relational operations, as well as parallel processing in the MapReduce framework.

[1]Cost estimation should be done using histograms on join attributes. A heuristic approximation is to estimate the join cost at each node N_i as the sum of the sizes of r_i and s_i, and choose range partitioning vectors to balance the sum of the sizes.

Telegram: @uni_k

www.konkur.in

22.4.1 Other Relational Operations

The evaluation of other relational operations also can be parallelized:

- **Selection**. Let the selection be $\sigma_\theta(r)$. Consider first the case where θ is of the form $a_i = v$, where a_i is an attribute and v is a value. If the relation r is partitioned on a_i, the selection proceeds at a single node. If θ is of the form $l \leq a_i \leq u$—that is, θ is a range selection—and the relation has been range-partitioned on a_i, then the selection proceeds at each node whose partition overlaps with the specified range of values. In all other cases, the selection proceeds in parallel at all the nodes.
- **Duplicate elimination**. Duplicates can be eliminated by sorting; either of the parallel sort techniques can be used, optimized to eliminate duplicates as soon as they appear during sorting. We can also parallelize duplicate elimination by partitioning the tuples (by either range or hash partitioning) and eliminating duplicates locally at each node.
- **Projection**. Projection without duplicate elimination can be performed as tuples are read in from disk in parallel. If duplicates are to be eliminated, either of the techniques just described can be used.
- **Aggregation**. Consider an aggregation operation. We can parallelize the operation by partitioning the relation on the grouping attributes, and then computing the aggregate values locally at each node. Either hash partitioning or range partitioning can be used. If the relation is already partitioned on the grouping attributes, the first step can be skipped.

  We can reduce the cost of transferring tuples during partitioning by partly computing aggregate values before partitioning, at least for the commonly used aggregate functions. Consider an aggregation operation on a relation r, using the **sum** aggregate function on attribute B, with grouping on attribute A. The system can perform the **sum** aggregation at each node N_i on those r tuples stored at N_i. This computation results in tuples with partial sums at each node; the result at N_i has one tuple for each A value present in r tuples stored at N_i, with the sum of the B values of those tuples. The system then partitions the result of the local aggregation on the grouping attribute A and performs the aggregation again (on tuples with the partial sums) at each node N_i to get the final result.

  As a result of this optimization, which is called **partial aggregation**, fewer tuples need to be sent to other nodes during partitioning. This idea can be extended easily to the **min** and **max** aggregate functions. Extensions to the **count** and **avg** aggregate functions are left for you to do in Exercise 22.2.

Skew handling for aggregation is easier than skew handling for joins, since the cost of aggregation is directly proportional to the input size. Usually, all that needs to be done is to use a good hash function to ensure the group-by attribute values are evenly distributed amongst the participating nodes. However, in some extreme cases, a few

Telegram: @uni_k

www.konkur.in

values occur very frequently in the group-by attributes, and hashing can lead to uneven distribution of values. When applicable, partial aggregation is very effective in avoiding skew in such situations. However, when partial aggregation is not applicable, skew can occur with aggregation.

Dynamic detection and handling of such skew can be done in some such cases: in case a node is found to be overloaded, some of the key values that are not yet processed by the node can be reassigned to another node, to balance the load. Such reassignment is greatly simplified if virtual-node partitioning is used; in that case, if a real node is found to be overloaded, some virtual nodes assigned to the overloaded real node, but not yet processed, are identified, and reassigned to other real nodes.

More information on skew handling for join and other operators may be found in the Further Reading section at the end of the chapter.

22.4.2 Map and Reduce Operations

Recall the MapReduce paradigm, described in Section 10.3, which is designed to ease the writing of parallel data processing programs.

Recall that the map() function provided by the programmer is invoked on each input record and emits zero or more output data items, which are then passed on to the reduce() function. Each data item output by a map() function consists of a record (*key, value*); we shall call the key as the **intermediate key**. In general, a map() function can emit multiple such records and since there are many input records, there are potentially many output records overall.

The MapReduce system takes all the records emitted by the map() functions, and groups them such that all records with a particular intermediate key are gathered together. The reduce() function provided by the programmer is then invoked for each intermediate key and iterates over a collection of all values associated with that key.

Note that the map function can be thought of as a generalization of the project operation: both process a single record at a time, but for a given input record the project operation generates a single output record, whereas the map function can output multiple records (including, as a special case, 0 records). Unlike the project operation, the output of a map function is usually intended to become the input of a reduce function; hence, the output of a map function has an associated key that serves as a group by attribute. Recall that the reduce function takes as input a collection of values and outputs a result; with most of the reduce functions commonly in use, the result is an aggregate computed on the input values, and the reduce function is then essentially a user-defined aggregation function.

MapReduce systems are designed for parallel processing of data. A key requirement for parallel processing is the ability to parallelize file input and output across multiple machines; otherwise, the single machine storing the data will become a bottleneck. Parallelization of file input and output can be done by using a distributed file system, such as the *Hadoop File System* (*HDFS*), discussed in Section 21.6, or by using a parallel/distributed storage system, discussed in Section 21.7. Recall that in such sys-

Telegram: @uni_k

www.konkur.in

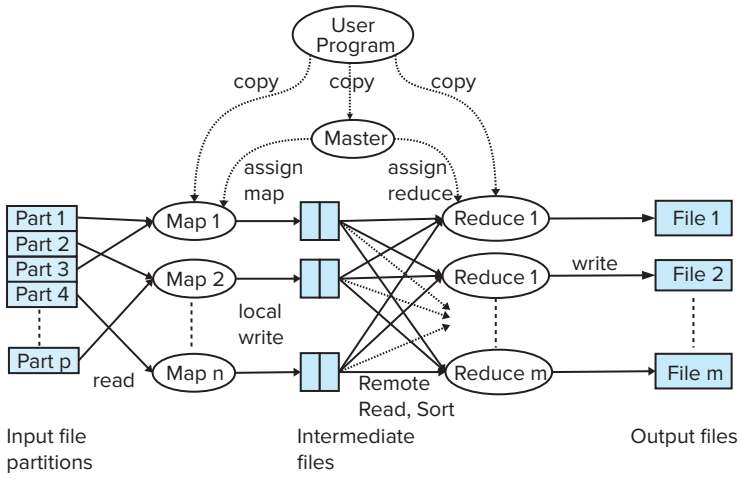


Figure 22.4 Parallel processing of MapReduce job.

tems, data are replicated (copied) across several (typically 3) machines, so that even if a few of the machines fail, the data are available from other machines that have copies of the data in the failed machine.

Conceptually, the map and reduce operations are parallelized in the same way that the relational operations project and aggregation are parallelized. Each node in the system has a number of concurrently executing **workers**, which are processes that execute map and reduce functions. The number of workers on one machine is often set to match the number of processor cores on the machine.

Parallel processing of MapReduce jobs is shown schematically in Figure 22.4. As shown in the figure, MapReduce systems split the input data into multiple pieces; the job of processing one such piece is called a **task**. Splitting can be done in units of files, and large files can be split into multiple parts. Tasks correspond to virtual nodes in our terminology, while workers correspond to real nodes. Note that with a multicore processor (as is standard today), MapReduce systems typically allocate one worker per core.

MapReduce systems also have a scheduler, which assigns tasks to workers.[2] Whenever a worker completes a task, it is assigned a new task, until all tasks have been assigned.

A key step between the map and reduce operations is the repartitioning of records output by the map step; these records are repartitioned based on their intermediate (reduce) key, such that all records with a particular key are assigned to the same reducer task. This could be done either by range-partitioning on the reduce key or by computing a hash function on the reduce key. In either case, the records are divided into multiple

[2]The scheduler is run on a dedicated node called the master node; the nodes that perform map() and reduce() tasks are called slave nodes in the Hadoop MapReduce terminology.

Telegram: @uni_k

www.konkur.in

partitions, each of which is called a reduce task. A scheduler assigns reduce tasks to workers.

This step is identical to the repartitioning done for parallelizing the relational aggregation operation, with records partitioned into a number of virtual nodes based on their group-by key.

To process the records in a particular reduce task, the records are sorted (or grouped) by the reduce key, so that all records with the same reduce-key value are brought together, and then the reduce() is executed on each group of reduce-key values.

The reduce tasks are executed in parallel by the workers. When a worker completes a reduce task, another task is assigned to it, until all reduce tasks have been completed. A reduce task may have multiple different reduce key values, but a particular call to the reduce() function is for a single reduce key; thus, the reduce() function is called for each key in the reduce task.

Tasks correspond to virtual nodes in the virtual-node partitioning scheme. There are far more tasks than there are nodes, and tasks are divided among the nodes. As discussed in Section 22.3.3, virtual-node partitioning reduces skew. Also note that as discussed in Section 22.4.1, skew can be reduced by partial aggregation, which corresponds to *combiners* in the MapReduce framework.

Further, MapReduce implementations typically also carry out dynamic detection and handling of skew, as discussed in Section 22.4.1.

Most MapReduce implementations include techniques to ensure that processing can be continued even if some nodes fail during query execution. Details are discussed further in Section 22.5.4.

22.5 Parallel Evaluation of Query Plans

As discussed in Section 22.1, there are two types of parallelism: intraoperation and interoperation. Until now in this chapter, we have focused on intraoperation parallelism. In this section, we consider execution plans for queries containing multiple operations.

We first consider how to exploit interoperator parallelism. We then consider a model of parallel query execution which breaks parallel query processing into two types of steps: partitioning of data using the *exchange operator*, and execution of operations on purely local data, without any data exchange. This model is surprisingly powerful and is widely used in parallel database implementations.

22.5.1 Interoperation Parallelism

There are two forms of interoperation parallelism: pipelined parallelism and independent parallelism. We first describe these forms of parallelism, assuming each operator runs on a single node without intraoperation parallelism.

Telegram: @uni_k

www.konkur.in

We then describe a model for parallel execution based on the exchange operator, in Section 22.5.2. Finally, in Section 22.5.3, we describe how a complete plan can be executed, combining all the forms of parallelism.

22.5.1.1 Pipelined Parallelism

Recall from Section 15.7.2 that in pipelining, the output tuples of one operation, A, are consumed by a second operation, B, even before the first operation has produced the entire set of tuples in its output. The major advantage of pipelined execution in a sequential evaluation is that we can carry out a sequence of such operations without writing any of the intermediate results to disk.

Parallel systems use pipelining primarily for the same reason that sequential systems do. However, pipelines are a source of parallelism as well, since it is possible to run operations A and B simultaneously on different nodes (or different cores of a single node), so that B consumes tuples in parallel with A producing them. This form of parallelism is called **pipelined parallelism**.

Pipelined parallelism is useful with a small number of nodes, but it does not scale up well. First, pipeline chains generally do not attain sufficient length to provide a high degree of parallelism. Second, it is not possible to pipeline relational operators that do not produce output until all inputs have been accessed, such as the set-difference operation. Third, only marginal speedup is obtained for the frequent cases in which one operator's execution cost is much higher than are those of the others.

All things considered, when the degree of parallelism is high, pipelining is a less important source of parallelism than partitioning. The real reason for using pipelining with parallel query processing is the same reason that pipelining is used with sequential query processing: namely, that pipelined executions can avoid writing intermediate results to disk.

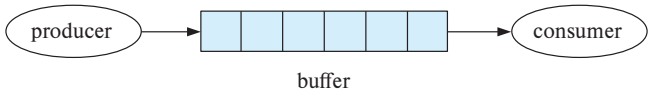
Pipelining in centralized databases was discussed in Section 15.7.2; as mentioned there, pipelining can be done using a demand-driven, or pull, model of computation, or using a producer-driven, or push, model of computation. The pull model is widely used in centralized database systems.

However, the push model is greatly preferred in parallel database systems, since, unlike the pull model, the push model allows both the producer and consumer to execute in parallel.

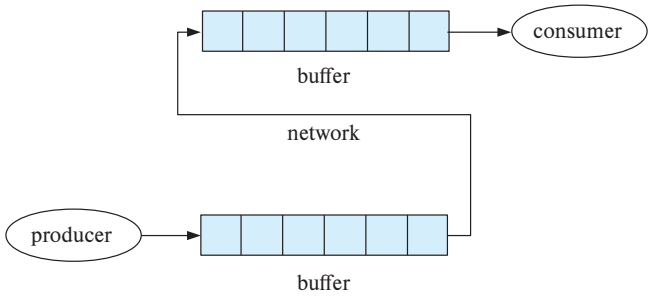
Unlike the pull model, the push model requires a buffer that can hold multiple tuples, between the producer and consumer; without such a buffer, the producer would stall as soon as it generates one tuple. Figure 22.5 shows a producer and consumer with a buffer in-between. If the producer and consumer are on the same node, as shown in Figure 22.5a, the buffer can be in shared memory. However, if the producer and consumer are in different nodes, as shown in Figure 22.5b, there will be two buffers: one at the producer node to collect tuples as they are produced, and another at the consumer node to collect them as they are sent across the network.

Telegram: @uni_k

www.konkur.in



(a) Producer-consumer in shared memory



(b) Producer-consumer across a network

Figure 22.5 Producer and consumer with buffer.

When sending tuples across a network, it makes sense to collect multiple tuples and send them as a single *batch*, rather than send tuples one at a time, since there is usually a very significant overhead per message. Batching greatly reduces this overhead.

If the producer and consumer are on the same node and can communicate via a shared memory buffer, mutual exclusion needs to be ensured when inserting tuples into, or fetching tuples from, the buffer. Mutual exclusion protocols have some overhead, which can be reduced by inserting/retrieving a batch of tuples at a time, instead of one tuple at a time.

Note that with the pull model, either the producer or the consumer, but not both, can be executing at a given time; while this avoids the contention on the shared buffer that arises with the use of the push model, it also prevents the producer and consumer from running concurrently.

22.5.1.2 Independent Parallelism

Operations in a query expression that do not depend on one another can be executed in parallel. This form of parallelism is called **independent parallelism**.

Consider the join $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$. One possible plan is to compute intermediate result $t_1 \leftarrow r_1 \bowtie r_2$ in parallel with intermediate result $t_2 \leftarrow r_3 \bowtie r_4$. Neither of these computations depends on each other, and hence they can be parallelized by independent parallelism. In other words, the execution of these two joins can be scheduled in parallel.

Telegram: @uni_k

www.konkur.in

When these two computations complete, we can compute:

$$t_1 \bowtie t_2$$

Note that computation of the above join depends on the results of the first two joins, hence it cannot be done using independent parallelism.

Like pipelined parallelism, independent parallelism does not provide a high degree of parallelism and is less useful in a highly parallel system, although it is useful with a lower degree of parallelism.

22.5.2 The Exchange Operator Model

The Volcano parallel database popularized a model of parallelization called the *exchange-operator* model. The exchange operation repartitions data in a specified way; data interchange between nodes is done only by the exchange operator. All other operations work on local data, just as they would in a centralized database system; the data may be available locally either because it is already present, or because of the execution of a preceding exchange operator.

The **exchange operator** has two components: a scheme for *partitioning* outgoing data, applied at each source node, and a scheme for *merging* incoming data, applied at each destination node. The operator is shown pictorially in Figure 22.6, with the partitioning scheme denoted as "Partition," and the merging scheme denoted as "Merge."

The exchange operator can *partition* data in one of several ways:

1. By hash partitioning on a specified set of attributes.
2. By range partitioning on a specified set of attributes.
3. By replicating the input data at all nodes, referred to as **broadcasting**.
4. By sending all data to a single node.

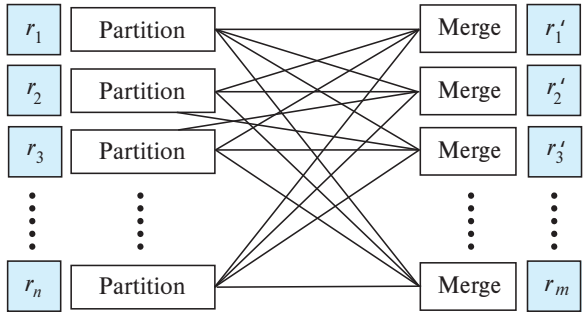


Figure 22.6 The exchange operator used for repartitioning.

Telegram: @uni_k

www.konkur.in

Broadcasting data to all nodes is required for operations such as the asymmetric fragment-and-replicate join. Sending all data to a single node is usually done as a final step of parallel query processing, to get partitioned results together at a single site.

Note also that the input to the exchange operator can be at a single site (referred to as **unpartitioned**), or already partitioned across multiple sites. Repartitioning of already partitioned data results in each destination node receiving data from multiple source nodes, as shown in Figure 22.6.

Each destination node *merges* the data items received from the source nodes. This merge step can store data in the order received (which may be nondeterministic, since it depends on the speeds of the machines and unpredictable network delays); such merging is called **random merge**.

On the other hand, if the input data from each source is sorted, the merge step can exploit the sort order by performing an **ordered merge**. Suppose, for example, nodes $N_1, \ldots, N_m$ first sort a relation locally, and then repartition the sorted relation using range partitioning. Each node performs an ordered merge operation on the tuples that it receives, to generate a sorted output locally.

Thus, the exchange operator performs the partitioning of data at the source nodes, as well the merging of data at the destination nodes.

All the parallel operator implementations we have seen so far can be modeled as a sequence of exchange operations, and local operators, at each node, that are completely unaware of parallelism.

- Range partitioning sort: can be implemented by an exchange operation that performs range partitioning, with random merge at the destination nodes, followed by a local sort operation at each destination node.
- Parallel external sort-merge: can be implemented by local sorting at the source nodes, followed by an exchange operation that performs range partitioning, along with ordered merging.
- Partitioned join: can be implemented by an exchange operation that performs the desired partitioning, followed by local join at each node.
- Asymmetric fragment-and-replicate join: can be implemented by an exchange operation that performs broadcast "partitioning" of the smaller relation, followed by a local join at each node.
- Symmetric fragment-and-replicate join: can be implemented by an exchange operation that partitions, and partially broadcasts each partition, followed by a local join at each node.
- Aggregation: can be implemented by an exchange operation that performs hash-partitioning on the grouping attributes, followed by a local aggregation operation at each node. The partial-aggregation optimization simply requires an extra local aggregation operation at each node, before the exchange operation.

Telegram: @uni_k

www.konkur.in

Other relational operations can be implemented similarly, by a sequence of local operations running in parallel at each node, interspersed with exchange operations.

As noted earlier, parallel execution where data are partitioned, and operations are executed locally at each node, is referred to as *data parallelism*. The use of the exchange operator model to implement data parallel execution has the major benefit of allowing existing database query engines to be used at each of the local nodes, without any significant code changes. As a result, the exchange-operator model of parallel execution is widely used in parallel database systems.

There are, however, some operator implementations that can benefit from being aware of the parallel nature of the system they are running on. For example, an indexed nested-loops join where the inner relation is indexed on a parallel data-store would require remote access for each index lookup; the index lookup operation is thus aware of the parallel nature of the underlying system. Similarly, in a shared-memory system it may make sense to have a hash table or index in shared-memory, which is accessed by multiple processors (this approach is discussed briefly in Section 22.6); the operations running on each processor are then aware of the parallel nature of the system.

As we discussed in Section 22.5.1.1, while the demand-driven (or pull) iterator model for pipelined execution of operators is widely used in centralized database engines, the push model is preferred for parallel execution of operators in a pipeline.

The exchange operator can be used to implement the push model between nodes in a parallel system, while allowing existing implementations of local operators to run using the pull model. To do so, at each source node of an exchange operator, the operator can pull multiple tuples from its input and create a batch of tuples destined for each destination node. The input may be computed by a local operation, whose implementation can use the demand-driven iterator model.

The exchange operator then sends batches of tuples to the destination nodes, where they are merged and kept in a buffer. The local operations can then consume the tuples in a demand-driven manner.

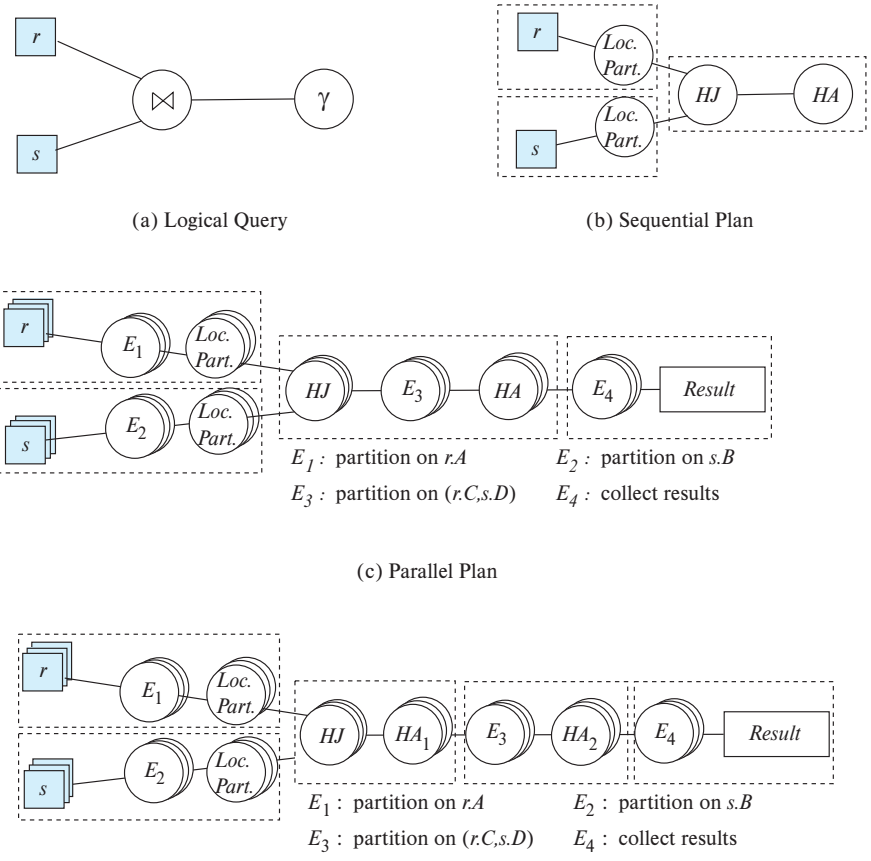
22.5.3 Putting It All Together

Figure 22.7 shows a query, along with a sequential and two alternative parallel query execution plans. The query, shown in Figure 22.7a, computes a join of two relations, r and s, and then computes an aggregate on the join result. Assume for concreteness that the query is ${}_{r.C,s.D}\gamma_{\text{sum}(s.E)}(r \bowtie_{r.A=s.B} s)$.

The sequential plan, shown in Figure 22.7b, uses a hash join (denoted as "HJ" in the figure), which executes in three separate stages. The first stage partitions the first input (r) locally on $r.A$; the second stage partitions the second input (s) locally on $s.B$; and the third stage computes the join of each of the corresponding partitions of r and s. The aggregate is computed using in-memory hash-aggregation, denoted by the operator HA; we assume that the number of groups is small enough that the hash table fits in memory.

Telegram: @uni_k

www.konkur.in



(a) Logical Query

(b) Sequential Plan

(c) Parallel Plan

(d) Parallel Plan with Partial Aggregation

Figure 22.7 Parallel query execution plans.

The dashed boxes in the figure show which steps run in a pipelined fashion. In the sequential plan, the read of the relation r is pipelined to the first partitioning stage of the sequential hash join; similarly, the read of relations s is pipelined to the second partitioning stage of the hash join. The third stage of the hash join pipelines its output tuples to the hash aggregation operator.

The parallel query evaluation plan, shown in Figure 22.7c, starts with r and s already partitioned, but not on the required join attributes.[3] The plan, therefore, uses the exchange operation E_1 to repartition r using attribute $r.A$; similarly, exchange operator E_2 repartitions s using $s.B$. Each node then uses hash join locally to compute the join

[3]Note the multiple boxes indicating a relation is stored in multiple nodes; similarly, multiple circles indicate that an operation is executed in parallel on multiple nodes.

Telegram: @uni_k

www.konkur.in

of its partition of r and s. Note that these partitions are not assumed to fit in memory, so they must be further partitioned by the first two stages of hash join; this local partitioning step is denoted as "Loc. Part." in the figure. The dashed boxes indicate that the output of the exchange operator can be pipelined to the local partitioning step. As in the sequential plan, there are two pipelined stages, one each for r and s. Note that exchange of tuples across nodes is done only by the exchange operator, and all other edges denote tuple flows within each node.

Subsequently, the hash join algorithm is executed in parallel at all participating nodes, and its output pipelined to the exchange operator E_3. This exchange operator repartitions its input on the pair of attributes $(r.C, s.D)$, which are the grouping attributes of the subsequent aggregation. At the receiving end of the exchange operator, tuples are pipelined to the hash aggregation operator. Note that the above steps all run together as a single pipelined stage, even though there is an exchange operator as part of the stage. Note that the local operators computing hash join and hash aggregate need not be aware of the parallel execution.

The results of the aggregates are then collected together at a central location by the final exchange operator E_4, to create the final result relation.

Figure 22.7d shows an alternative plan that performs partial aggregation on the results of the hash join, before partitioning the results. The partial aggregation is computed locally at each node by the operator HA_1. Since no tuple is output by the partial aggregation operator until all its input is consumed, the pipelined stage contains only the local hash join and hash aggregation operators. The subsequent exchange operator E_3 which partitions its input on $(r.C, s.D)$ is part of a subsequent pipelined stage along with the hash aggregation operation HA_2 which computes the final aggregate values. As before, the exchange operator E_4 collects the results at a centralized location.

The above example shows how pipelined execution can be performed across nodes, as well as within nodes, and further how it can be done along with intra-operator parallel execution. The example also shows that some pipelined stages depend on the output of earlier pipelined stages; therefore their execution can start only after the previous step finishes. On the other hand, the initial exchange and partitioning of r and s occur in pipelined stages that are independent of each other; such independent stages can be scheduled concurrently, that is, at the same time, if desired.

To execute a parallel plan such as the one in our example, the different pipelined stages have to be scheduled for execution, in an order that ensures inter-stage dependencies are met. When executing a particular stage, the system must decide how many nodes an operation should be executed on. These decisions are usually made as part of the scheduling phase, before query execution starts.

22.5.4 Fault Tolerance in Query Plans

Parallel processing of queries across a moderate number of nodes, for example, hundreds of nodes, can be done without worrying about fault tolerance. If a fault occurs, the query is rerun, after removing any failed nodes from the system (replication of data

Telegram: @uni_k

www.konkur.in

at the storage layer ensures that data continues to be available even in the event of a failure). However, this simple solution does not work well when operating at the scale of thousands or tens of thousands of nodes: if a query runs for several hours, there is a significant chance that there will be a failure while the query is being executed. If the query is restarted, there is a significant chance of another failure while it is executing, which is obviously an undesirable situation.

To deal with this problem, the query processing system should ideally just be able to redo the actions of a failed node, without redoing the rest of the computation.

Implementations of MapReduce that are designed to work at a massively parallel scale can be made fault tolerant as follows:

1. Each map operation executed at each node writes its output to local files.
2. The next operation, which is a reduce operation, executes at each node; the operation execution at a node reads data from the files stored at multiple nodes, collects the data, and starts processing the data only after it has got all its required data.
3. The reduce operation writes its output to a distributed file system (or distributed storage system) that replicates data, so that the data would be available even in the event of a failure.

Let us now examine the reason why things are done as above. First, if a particular map node fails, the work done at that node can be redone at a backup node; the work done at other map nodes is not affected. Work is not carried out by reduce nodes until all the required data has been fetched; the failure of a map node just means the reduce nodes fetch data from the backup map nodes. There is certainly a delay while the backup node does its work, but there is no need to repeat the entire computation.

Further, once a reduce node has finished its work, its output goes to replicated storage to ensure it is not lost even if a data storage node fails. This means that if a reduce node fails before it completes its work, it will have to be reexecuted at a backup node; other reduce nodes are not affected. Once a reduce node has finished its work, there is no need to reexecute it.

Note that it is possible to store the output of a map node in a replicated storage system. However, this increases the execution cost significantly, and hence map output is stored in local storage, even at the risk of having to reexecute the work done by a map node in case it fails before all the reduce nodes have fetched the data that they require from that map node.

It is also worth noting that sometimes nodes do not completely fail, but run very slowly; such nodes are called *straggler* nodes. Even a single straggler node can delay all the nodes in the next step (if there is a following step), or delay task completion (if it is in the last step). Straggler nodes can be dealt with by treating them similar to failed nodes, and reexecuting their tasks on other nodes (the original task on the straggler node can also be allowed to continue, in case it finishes first). Such reexecution to deal with

Telegram: @uni_k

www.konkur.in

stragglers has been found to significantly improve time to completion of MapReduce tasks.

While the above scheme for fault tolerance is quite effective, there is an overhead that must be noted: a reduce stage cannot perform any work until the previous map stage has finished;[4] and if multiple map and reduce steps are executed, the next map stage cannot perform any work until the preceding reduce stage has finished. In particular, this means that pipelining of data between stages cannot be supported; data are always materialized before it is sent to the next stage. Materialization carries a significant overhead, which can slow down computation.

Apache Spark uses an abstraction called *Resilient Distributed Datasets* (RDDs) to implement fault tolerance. As we have seen in Section 10.4.2, RDDs can be viewed as collections, and Spark supports algebraic operations that take RDDs as input, and generate RDDs as output. Spark keeps track of the operations used to create an RDD. In case of failures that result in loss of an RDD, the operations used to create the RDD can be reexecuted to regenerate the RDD. However, this may be time-consuming, so Spark also supports replication to reduce the chance of data loss, as well as storing of local copies of data when a shuffle (exchange) step is executed, to allow reexecution to be restricted to computation that was performed on failed nodes.

There has been a good deal of research on how to allow pipelining of data, while not requiring query execution to restart from the beginning in case of a single failure. Such schemes typically require nodes to track what data they have received from each source node. In the event of a source node failure, the work of the source node is redone on a backup node, which can result in some tuples that were received earlier being received again. Tracking the data received earlier is important to ensure duplicate tuples are detected and eliminated by the receiving node. The above ideas can also be used to implement fault tolerance for other algebraic operations, such as joins. In particular, if we use the exchange operator with data parallelism, fault tolerance can be implemented as an extension of the exchange operator.

References to more information on fault tolerant pipelining based on extensions of the exchange operator, as well as on fault tolerance schemes used in MapReduce and in Apache Spark, may be found in the Further Reading section at the end of the chapter.

22.6 Query Processing on Shared-Memory Architectures

Parallel algorithms designed for shared-nothing architectures can be used in shared-memory architectures. Each processor can be treated as having its own partition of memory, and we can ignore the fact that the processors have a common shared-

[4]Once a map node finishes its tasks, redistribution of results from that node to the reduce nodes can start even if other map nodes are still active; but the actual computation at the reduce node cannot start until all map tasks have completed and all map results redistributed.

Telegram: @uni_k

www.konkur.in

memory. However, execution can be optimized significantly by exploiting the fast access to shared-memory from any of the processors.

Before we study optimizations that exploit shared-memory, we note that while many large-scale systems can execute on a single shared-memory system, the largest-scale systems today are typically implemented using a hierarchical architecture, with a shared-nothing architecture at the outer level, but with each node having a shared-memory architecture locally, as discussed in Section 20.4.8. The techniques we have studied so far for storing, indexing, and querying data in shared-nothing architectures are used to divide up storage, indexing, and query processing tasks among the different nodes in the system. Each node is a shared-memory parallel system, which uses parallel query processing techniques to execute the query processing tasks assigned to it. The optimizations we describe in this section can thus be used locally, at each node.

Parallel processing in a shared memory system is typically done by using threads, rather than separate processes. A **thread** is an execution stream that shares its entire memory[5] with other threads. Multiple threads can be started up, and the operating system schedules threads on available processors.

We list below some optimizations that can be applied when parallel algorithms that we saw earlier are executed in a shared memory system.

1. If we use asymmetric fragment-and-replicate join, the smaller relation need not be replicated to each processor. Instead, only one copy needs to be stored in shared memory, which can be accessed by all the processors. This optimization is particularly useful if there are a large number of processors in the shared-memory system.

2. Skew is a significant problem in parallel systems, and it becomes worse as the number of processors grows. Handing off work from an overloaded node to an underloaded node is expensive in a shared-nothing system since it involves network traffic. In contrast, in a shared memory system, data assigned to a processor can be easily accessed from another processor.

   To address skew in a shared-memory system, a good option is to use virtual-node partitioning, which allows work to be redistributed in order to balance load. Such redistribution could be done when a processor is found to be overloaded. Alternatively, whenever a processor finds that it has finished processing all the virtual nodes assigned to it, it can find other processors that still have virtual nodes left to be processed, and take over some of those tasks; as mentioned in Section 22.3.3, this approach is called *work stealing*. Note that such an approach to avoiding skew would be much more expensive in a shared-nothing environment since a significant amount of data movement would be involved, unlike in the shared-memory case.

[5]Technically, in operating-system terminology, its address space.

Telegram: @uni_k

www.konkur.in

3. Hash join can be executed in two distinct ways.

   a. The first option is to partition both relations to each processor and then compute the joins of the partitions, in a manner similar to shared-nothing hash join. Each partition must be small enough that the hash index on a build-relation partition fits in the part of shared memory allocated to each processor.

   b. The second option is to partition the relations into fewer pieces, such that the hash index on a build-relation partition fits into common shared memory, rather than a fraction of the shared memory. The construction of the in-memory index, as well as probing of the index, must now be done in parallel by all the processors.

      Parallelizing the probe phase is relatively easy, since each processor can work on some partition of the probe relation. In fact it makes sense to use the virtual node approach and partition the probe relation into many small pieces (sometimes called "morsels"), and have processors process a morsel at a time. When a processor is done with a morsel, it finds an unprocessed morsel and works on it, until there are no morsels left to be processed.

      Parallelizing the construction of the shared hash index is more complicated, since multiple processors may attempt to update the same part of the hash index. Using locks is an option, but there are overheads due to locking. Techniques based on lock-free data structures can be used to construct the hash index in parallel.

References to more details on how to parallelize join implementations in shared memory may be found in the Further Reading section at the end of the chapter.

Algorithms designed for shared-memory systems must take into account the fact that in today's processors, memory is divided into multiple memory banks, with each bank directly linked to some processor. The cost of accessing memory from a given processor is less if the memory is directly linked to the processor, and is more if it is linked to a different processor. Such memory systems are said to have a *Non-Uniform Memory Access* or *NUMA* architecture.

To get the best performance, algorithms must be *NUMA-aware*; that is, they must be designed to ensure that data accessed by a thread running on a particular processor is, as far as possible, stored in memory local to that processor. Operating systems support this task in two ways:

1. Each thread is scheduled, as far as possible, on the same processor core, every time it is executed.
2. When a thread requests memory from the operating system memory manager, the operating system allocates memory that is local to that processor core.

Note that the techniques for making the best use of shared memory are complementary to techniques that make the best use of processor caches, including cache-

Telegram: @uni_k

www.konkur.in

conscious index structures (which we saw in Section 14.4.7) and cache-conscious algorithms for processing relational operators.

But in addition, since each processor core has its own cache, it is possible for a cache to have an old value that was subsequently updated on another processor core. Thus, query processing algorithms that update shared data structures should be careful to ensure that there are no bugs due to the use of outdated values, and due to race conditions on updating the same memory location from two processor cores. Locking and fence instructions to ensure cache consistency (Section 20.4.5) are used in combination to implement updates to shared data structures.

The form of parallelism we have studied so far allows each processor to execute its own code independently of other processors. However, some parallel systems support a different form of parallelism, called **Single Instruction Multiple Data** (**SIMD**). With SIMD parallelism, the same instruction is executed on each of multiple data items, which are typically elements of an array. SIMD architectures became widely used in graphics processing units (GPUs), which were initially used for speeding up processing of computer graphics tasks. However, more recently, GPU chips have been used for parallelizing a variety of other tasks, one of which is parallel processing of relational operations using the SIMD support provided by GPUs. Intel's Xeon Phi coprocessor supports not only multiple cores in a single chip, but also several SIMD instructions that can operate in parallel on multiple words. There has been a good deal of research on how to process relational operations in parallel on such SIMD architectures; references to more information on this topic may be found in the bibliographic notes for this chapter, available online.

22.7 Query Optimization for Parallel Execution

Query optimizers for parallel query evaluation are more complicated than query optimizers for sequential query evaluation. First, the space of plan alternatives can be much larger than that for sequential plans. In particular, in a parallel setting, we need to take into account the different possible ways of partitioning the inputs and intermediate results, since different partitioning schemes can lead to different query execution costs, which is not an issue for a sequential plan.

Second, the cost models are more complicated, since the cost of partitioning must be taken into account, and issues such as skew and resource contention must be addressed.

22.7.1 Parallel Query Plan Space

As we have seen in Section 15.1, a sequential query plan can be expressed as an algebraic expression tree, where each node is a physical operator, such as a sort operator, a hash join operator, a merge-join operator, etc. Such a plan may be further annotated with instructions on what operations are to be pipelined and what intermediate results are to be materialized, as we saw in Section 15.7.

Telegram: @uni_k

www.konkur.in

In addition to the above, a parallel query plan must specify

- How to parallelize each operation, including deciding what algorithm to use, and how to partition the inputs and intermediate results. Exchange operators can be used to partition inputs as well as intermediate results and final results.
- How the plan is to be **scheduled**; specifically:
  - How many nodes to use for each operation.
  - What operations to pipeline within the same node, or across different nodes.
  - What operations to execute sequentially, one after the other.
  - What operations to execute independently in parallel.

As an example of the partitioning decision, a join $r \bowtie_{r.A=s.A \wedge r.B=s.B} s$ can be parallelized by partitioning r and s on the attributes $r.A$ and $s.A$ alone, or on the attributes $r.B$ and $s.B$ alone, or on $(r.A, r.B)$ and $(s.A, s.B)$. The last option is likely to be the best if we consider only this join, since it minimizes the chances of skew which can arise if the cardinalities of $r.A$, $r.B$, $s.A$ or $s.B$ are low.

But consider now the query ${}_{r.A}\gamma_{\mathbf{sum}(s.C)}(r \bowtie_{r.A=s.A \wedge r.B=s.B} s)$. If we perform the join by partitioning on $(r.A, r.B)$ (and $(s.A, s.B)$), we would then need to repartition the join result by $r.A$ to compute the aggregate. On the other hand, if we performed the join by partitioning on r and s on $r.A$ and $s.A$ respectively, both the join and the aggregate can be computed without any repartitioning, which could reduce the cost. This option is particularly likely to be a good option if $r.A$ and $s.A$ have high cardinality and few duplicates, since the chance of skew is less in this case.

Thus, the optimizer has to consider a larger space of alternatives, taking partitioning into account. References with more details about how to implement query optimization for parallel query processing systems, taking partitioning alternatives into account, may be found in the Further Reading section at the end of the chapter.

22.7.2 Cost of Parallel Query Evaluation

The cost of a sequential query plan is typically estimated based on the total resource consumption of the plan, adding up the CPU and I/O costs of the operators in a query plan. The *resource consumption* cost model can also be used in a parallel system, additionally taking into account the network cost, and adding it along with the other costs. As discussed in Section 15.2, even in a sequential system, the resource consumption cost model does not guarantee minimization of the execution time of an individual query. There are other cost models that are better at modeling the time to complete a query; however, the resource consumption cost model has the benefit of reducing the cost of query optimization, and is thus widely used. We return to the issue of other cost models later in the section.

Telegram: @uni_k

www.konkur.in

We now study how the cost of a parallel query plan can be estimated based on the resource consumption model. If a query plan is data parallel, then each operation, other than the exchange operation, runs locally; the cost of such operations can be estimated using techniques we have seen earlier in Chapter 15, if we assume that the input relations are uniformly partitioned across n nodes, with each node receiving $1/n$th of the overall input.

The cost of the exchange operation can be estimated based on the network topology, the amount of data transferred, and the network bandwidth; as before it is assumed that each node is equally loaded during the exchange operation. The cost of a query plan under the *resource-consumption* model can then be found by adding up the costs of the individual operations.

However, in a parallel system, two plans with the same resource consumption may have significantly different time to completion. A **response-time cost model** is a cost model that attempts to better estimate the time to completion of a query. If a particular operation is able to perform I/O operations in parallel with CPU execution, the response time would be better modeled as *max*(CPU cost, I/O cost), rather than the resource consumption cost model of (CPU cost + I/O cost). Similarly, if two operations o_1 and o_2 are in a pipeline on a single node, and their CPU and I/O costs are c_1, io_1 and c_2, io_2 respectively, then their response time cost would be $max(c_1 + c_2, io_1 + io_2)$. Similarly, if operations o_1 and o_2 are executed sequentially, then their response time cost would be $max(c_1, io_1) + max(c_2, io_2)$.

When executing operations in parallel across multiple nodes, the response time cost model would have to take into account:

- **Start-up costs** for initiating an operation at multiple nodes.
- **Skew** in the distribution of work among the nodes, with some nodes getting a larger number of tuples than others, and thus taking longer to complete. It is the time to completion of the slowest node that determines the time to completion of the operation.

Thus, any skew in the distribution of the work across nodes greatly affects performance.

Estimating the time to completion of the slowest node due to skew is not an easy task since it is highly data dependent. However, statistics such as number of distinct values of partitioning attributes, histograms on the distribution of values of partitioning attributes, and counts of most frequent values can be used to estimate the potential for skew. Partitioning algorithms that can detect and minimize the effect of skew, such as those discussed in Section 21.3, are important to minimize skew.

22.7.3 Choosing a Parallel Query Plan

The number of parallel evaluation plans from which to choose is much larger than the number of sequential evaluation plans. Optimization of parallel queries by considering all alternatives is therefore much more expensive than optimization of sequential

Telegram: @uni_k

www.konkur.in

queries. Hence, we usually adopt heuristic approaches to reduce the number of parallel execution plans that we have to consider. We describe two popular approaches here.

1. A simple approach is to choose the most efficient sequential evaluation plan, and then to choose the optimal way to parallelize the operations in that evaluation plan. When choosing a sequential plan, the optimizer may use a basic sequential cost model; or it may use a simple cost model that takes some aspects of parallel execution into account but does not consider issues such as partitioning or scheduling. This option allows an existing sequential query optimizer to be used with minimal changes for the first step.

   Next, the optimizer decides how to create an optimal parallel plan corresponding to the chosen sequential plan. At this point, choices of what partitioning techniques to use and how to schedule operators can be made in a cost-based manner.

   The chosen sequential plan may not be optimal in the parallel context, since the exact cost formulae for parallel execution were not used when choosing it; nevertheless, the approach works reasonably well in many cases.

2. A more principled approach is to find the best plan, assuming that each operation is executed in parallel across all the nodes (operations with very small inputs may be executed on fewer nodes). Scheduling of independent operations in parallel on different nodes is not considered at this stage.

   Partitioning of inputs and intermediate results is taken into consideration when estimating the cost of a query plan. Existing techniques for query optimization have been extended by considering partitioning as a physical property, in addition to physical properties such as sort orders that are already taken into account when choosing a sequential query plan. Just as sort operators are added to a query plan to get a desired sort order, exchange operators are added to get the desired partitioning property. The cost model used in practice is typically a resource consumption model, which we saw earlier. Although response-time cost models offer better estimates of query execution time, the cost of query optimization is higher when using a response-time cost model compared to the cost of optimization when using a resource-consumption cost model. References providing more information on the response-time cost model may be found in the Further Reading section at the end of the chapter.

Yet another dimension of optimization is the design of physical-storage organization to speed up queries. For example, a relation can be stored partitioned on any of several different attributes, and it may even be replicated and replicas can be stored partitioned on different attributes. For example, a relation $r(A, B, C)$ could be stored partitioned on A, and a replica could be partitioned on B. The query optimizer chooses the replica that is best suited for the query. The optimal physical organization differs

Telegram: @uni_k

www.konkur.in

for different queries. The database administrator must choose a physical organization that appears to be good for the expected mix of database queries.

22.7.4 Colocation of Data

Even with parallel data storage and parallel processing of operations, the execution time of some queries can be too slow for the needs of some applications. In particular, queries that access small amounts of data stored at multiple nodes may run quite slowly when executed in parallel, as compared to the execution of the same query on a single node, if all the data were available at that node. There are many applications that need such queries to return answers with very low latency.

An important technique to speed up such queries is to colocate data that a query accesses in a single node. For example, suppose an application needs to access student information, along with information about courses taken by the student. Then, the *student* relation may be partitioned on the *ID*, and the *takes* relation also partitioned in exactly the same manner on *ID*. Tuples in the *course* relation, which is a small relation, may be replicated to all nodes. With such a partitioning, any query involving these three relations that retrieves data for a single *ID* can be answered locally at a single node. The query processing engine just detects which node is responsible for that *ID* and sends the query to that node, where it is executed locally.

Colocation of tuples from different relations is supported by many data storage systems. If the data storage system does not natively support colocation, an alternative is to create an object containing related tuples that share a key (e.g., *student* and *takes* records corresponding to a particular *ID*), and store it in the data storage system with the associated key (*ID*, in our example).

The colocation technique, however, does not work directly if different queries need a relation partitioned in different ways. For example, if the goal is to find all students who have taken a particular course section, the *takes* relation needs to be partitioned on the section information (*course_id*, *year*, *semester*, *sec_id*) instead of being partitioned on *ID*.

A simple technique to handle this situation is to allow multiple copies of a relation, partitioned on different attributes. These copies can be modeled as indices on the relation, partitioned on different attributes; when tuples in the relation are updated, so are the copies, to keep them consistent. In our example, one copy of *takes* can be partitioned on *ID* to be colocated with the *student* partitions, while another copy is partitioned on (*course_id*, *year*, *semester*, *sec_id*) to be colocated with the *section* relation.

Colocation helps optimize queries that compute joins between two relations; it can extend to three or more relations if either the remaining relations are replicated, or if all relations share a common set of join attributes. In the latter case, all tuples that would join together can be colocated by partitioning on the common join attributes. In either case, the join can be computed locally at a single node, if the query only wants the results for a specific value of the join attribute, as we saw earlier. However, not all join

Telegram: @uni_k

www.konkur.in

queries are amenable to evaluation at a single node by using colocation. Materialized views, which we discuss next, offer a more general alternative.

22.7.5 Parallel Maintenance of Materialized Views

Materialized views, which we saw in Section 4.2.3 for speeding up queries in centralized databases, can also be used to speed up queries in parallel databases. Materialized views need to be maintained when the database is updated, as we saw in Section 16.5. Materialized views in a parallel database can have a very large amount of data, and must, therefore, be stored partitioned across multiple nodes.

As in the centralized case, materialized views in a parallel database speed up query answering at the cost of the overhead of view maintenance at the time of processing updates.

We consider first a very simple case of materialized views. It is often useful to store an extra copy of a relation, partitioned on different attributes, to speed up answering of some queries. Such a repartitioning can be considered a very simple case of a materialized view; view maintenance for such a view is straightforward, just requiring updates to be sent to the appropriate partition.

Indices can be considered a form of materialized views. Recall from Section 21.5 how parallel indices are maintained. Consider the case of maintenance of a parallel secondary index on an attribute B of a relation $r(A, B, C)$, with primary key A. The secondary index would be sorted on attribute B and would include unique key A; assume it also includes attribute C. Suppose now that attribute B of a tuple $(a1, b1, c1)$ is updated from $b1$ to $b2$. This update results in two updates to the secondary index: delete the index entry with key $(b1, a1, c1)$, and add an entry $(b2, a1, c1)$. Since the secondary index is itself partitioned, these two updates need to be sent to the appropriate partition, based on the unique key attributes (B, A).

In some cases, materialized view maintenance can be done by partitioning followed by local view maintenance. Consider a view that groups *takes* tuples by (*course_id*, *year*, *semester*, *sec_id*), and then counts the number of students who have taken that course section. Such a materialized view would be stored partitioned on the grouping attributes (*course_id*, *year*, *semester*, *sec_id*). It can be computed by maintaining a copy of the *takes* relation partitioned on (*course_id*, *year*, *semester*, *sec_id*), and materializing the aggregates locally at each node. When there is an update, say an insert or delete to the *takes* relation, that update must be propagated to the appropriate node based on the partitioning chosen above. The materialized aggregate can be maintained locally at each node, as updates are received for the set of local tuples of the *takes* relation.

For more complex views, materialized view maintenance cannot be done by a single step of partitioning and local view maintenance. We consider a more general approach next.

First, consider an operator o, whose result is materialized, and an update (insert or delete) to one of the input relations of o that requires maintenance of the materialized result of o. Suppose the execution of operator o is parallelized using the exchange op-

Telegram: @uni_k

www.konkur.in

erator model (Section 22.5.2) where the inputs are partitioned, and then operators are executed at each node on data made available locally by (re)partitioning. To support materialized view maintenance of the result of o, we materialize the output of o at each node and additionally materialize (store) the input partitions sent to the node when the materialized view result is initially computed.

Now, when there is an update (insert or delete) to an input to o, we send the update to the appropriate node using the same partition function used during the initial computation of o. Consider a node that has received such an update. Now, the maintenance of the locally materialized result at the node can be done using standard (nonparallel) view maintenance techniques using only locally available data.

Note that as we saw in Section 22.5.2, a variety of operations can be parallelized using the exchange operator model, and hence the preceding scheme provides a parallel view maintenance technique for all such operators.

Next, consider a query with multiple operators. Such a query can be parallelized using the exchange operator model. The exchange operators repartition data between nodes, and each node computing (possibly multiple) operations using data made available locally by the exchange operators, as we saw in Section 22.5.2.

We can materialize the inputs and results at each node. Now, when there is a change to an input at a node, we use standard view maintenance techniques locally to find the change to the materialized result, say v, at that node. If that result v is the final output of the query, we are done. Otherwise, there must be an exchange operator above it; we use the exchange operator to route the updates (inserts or delete) to v to the next operator. That operator in turn computes the change to its result, and propagates it further if required, until we get to the root of the original materialized view.

The issue of consistency of materialized views in the face of concurrent updates to the underlying relations is addressed in Section 23.6.3.

22.8 Parallel Processing of Streaming Data

We saw several applications of streaming data in Section 10.5. Many of the streaming data applications that we saw in that section, such as network monitoring or stock market applications, have very high rates of tuple arrival. Incoming tuples cannot be processed by a single computer, and parallel processing is essential for such systems. Streaming data systems apply a variety of operations on incoming data. We now see how some of these operations can be executed in parallel.

Parallelism is essential at all stages of query processing, starting with the entry of tuples from the sources. Thus, a parallel stream processing system needs to support a large number of entry points for data.

For example, a system that is monitoring queries posed on a search engine such as Google or Bing search has to keep up with a very high rate of queries. Search engines have a large number of machines across which user queries are distributed and executed. Each of these machines becomes a source for the query stream. The stream

Telegram: @uni_k

www.konkur.in

processing system must have multiple entry points for the data, which receive data from the original sources and route them within the stream processing system.

Processing of data must be done by routing tuples from producers to consumers. We discuss routing of tuples in Section 22.8.1. Parallel processing of stream operations is discussed in Section 22.8.2, while fault tolerance is discussed in Section 22.8.3.

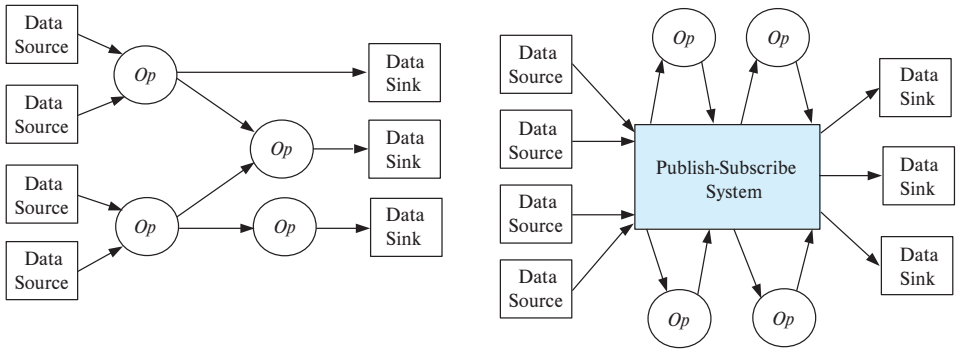
It is also worth noting that many applications that perform real-time analytics on streaming data also need to store the data and analyze it in other ways subsequently. Thus, many systems duplicate incoming data streams, sending one copy to a storage system for subsequent analysis and sending the other copy to a streaming data system; such an architecture is called the **lambda** architecture: the Greek symbol λ is used to pictorially denote that incoming data are forked into two copies, sent to two different systems.

While the lambda architecture allows streaming systems to be built quickly, it also results in duplication of effort: programmers need to write code to store and query the data in the format/language supported by a database, as well as to query the data in the language supported by a streaming data system. More recently, there have been efforts to perform stream processing as well as query processing on stored data within the same system to avoid this duplication.

22.8.1 Routing of Tuples

Since processing of data typically involves multiple operators, routing of data to operators is an important task. We first consider the logical structure of such routing, and address the physical structure, which takes parallel processing into account, later.

The *logical routing* of tuples is done by creating a directed acyclic graph (DAG) with operators as nodes. Edges between nodes define the flow of tuples. Each tuple output by an operator is sent along all the out-edges of the operator, to the consuming operators. Each operator receives tuples from all its in-edges. Figure 22.8a depicts the logical



(a) DAG representation of streaming data flow

(b) Publish-subscribe representation of streaming data flow

Figure 22.8 Routing of streams using DAG and publish-subscribe representations.

Telegram: @uni_k

www.konkur.in

routing of stream tuples through a DAG structure. Operation nodes are denoted as "Op" nodes in the figure. The entry points to the stream processing system are the data source nodes of the DAG; these nodes consume tuples from the stream sources and inject them into the stream processing system. The exit points of the stream processing system are data sink nodes; tuples exiting the system through a data sink may be stored in a data store or file system or may be output in some other manner.

One way of implementing a stream processing system is by specifying the graph as part of the system configuration, which is read when the system starts processing tuples and is then used to route tuples. The Apache Storm stream processing system is an example of a system that uses a configuration file to define the graph, which is called a *topology* in the Storm system. (Data source nodes are called *spouts* in the Storm system, while operator nodes are called *bolts*, and edges connect these nodes.)

An alternative way of creating such a routing graph is by using **publish-subscribe** systems. A publish-subscribe system allows publication of documents or other forms of data, with an associated topic. Subscribers correspondingly subscribe to specified topics. Whenever a document is published to a particular topic, a copy of the document is sent to all subscribers who have subscribed to that topic. Publish-subscribe systems are also referred to as **pub-sub** systems for short.

When a publish-subscribe system is used for routing tuples in a stream processing system, tuples are considered documents, and each tuple is tagged with a topic. The entry points to the system conceptually "publish" tuples, each with an associated topic. Operators subscribe to one or more topics; the system routes all tuples with a specific topic to all subscribers of that topic. Operators can also publish their outputs back to the publish-subscribe system, with an associated topic.

A major benefit of the publish-subscribe approach is that operators can be added to the system, or removed from it, with relative ease. Figure 22.8b depicts the routing of tuples using a publish-subscribe representation. Each data source is assigned a unique topic name; the output of each operator is also assigned a unique topic name. Each operator subscribes to the topics of its inputs and publishes to the topics corresponding to its output. Data sources publish to their associated topic, while data sinks subscribe to the topics of the operators whose output goes to the sink.

The Apache Kafka system uses the publish-subscribe model to manage routing of tuples in streams. In the Kafka system, tuples published for a topic are retained for a specified period of time (called the retention period), even if there is currently no subscriber for the topic. Subscribers usually process tuples at the earliest possible time, but in case processing is delayed or temporarily stopped due to failures, the tuples are still available for processing until the retention time expires.

Many streaming data systems, such as Google's Millwheel, and the Muppet stream processing system, use the term **stream** in place of the term *topic*. In such systems, streams are assigned names; operators can publish tuples to a stream, or subscribe to a stream, based on the name.

We now consider the *physical routing* of tuples. Regardless of the model used above, each logical operator must have multiple physical instances running in parallel on dif-

Telegram: @uni_k

www.konkur.in

ferent nodes. Incoming tuples for a logical operator must be routed to the appropriate physical instance(s) of the operator. A partitioning function is used to determine which tuple goes to which instance of the operator.

In the context of a publish-subscribe system, each topic can be thought of as a separate logical operator that accepts tuples and passes them on to all subscribers of the topic. Since there may be a very large number of tuples for a given topic, they must be processed in parallel across multiple nodes in a parallel publish-subscribe system. In the Kafka system, for example, a topic is divided into multiple partitions, called a **topic-partition**; each tuple for that topic is sent to only one of the topic-partitions. Kafka allows a partition key to be attached to each tuple, and ensures that tuples with the same key are delivered to the same partition.

To allow processing by consumers, Kafka allows consumer operators register with a specified "consumer group." The consumer group corresponds to a logical operator, while the individual consumers correspond to physical instances of the logical operator that run in parallel. Each tuple of a topic is sent to only one consumer in the consumer group. More precisely, all tuples in a particular topic-partition are sent to a single consumer in a consumer group; however, tuples from multiple partitions may be sent to the same consumer, leading to a many-to-one relationship from partitions to consumers.

Kafka is used in many streaming data processing implementations for routing tuples. Kafka Streams provides a client library supporting algebraic operations on streams, which can be used to build streaming applications on top of the Kafka publish-subscribe system.

22.8.2 Parallel Processing of Stream Operations

For standard relational operations, the techniques that we have seen for parallel evaluation of the operations can be used with streaming data.

Some of these, such as selection and projection operations, can be done in parallel on different tuples. Others, such as grouping, have to bring all tuples of a group together to one machine.[6] When grouping is done with aggregation, optimizations such as pre-aggregation can be used to reduce the data transferred, but information about the tuples in a group must still be delivered to a single machine.

Windowing is an important operation in streaming data systems. Recall from Section 10.5.2.1 that incoming data are divided into windows, typically based on timestamps (windows can also be defined based on the number of tuples). Windowing is often combined with grouping/aggregation, with aggregates computed on groups of tuples within a window. The use of windows ensures that once the system can determine that new tuples will no longer belong to a particular window, aggregates for that window can be output. For example, suppose a window is based on time, say with each 5 minutes defining a window; once the system determines that future tuples will have a

[6]When grouping is combined with windowing (Section 10.5.2.1), a group contains all tuples in a window that have the same values for their grouping attributes.

Telegram: @uni_k

www.konkur.in

timestamp larger than the end of a particular window, aggregates on that window can be output. Unlike grouping, windows can overlap each other.

When windowing and grouping are used together to compute aggregates, if there are overlapping windows, it is best to partition on just the grouping attributes. Otherwise, tuples which belong to multiple windows would have to be sent to multiple windows, an overhead that is avoided by partitioning on only the grouping attributes.

Many streaming systems allow users to create their own operators. It is important to be able to parallelize user-defined operators by allowing multiple instances of the operator to run concurrently. Such systems typically require each tuple to have an associated key, and all tuples with a particular key are sent to a particular instance of the operator. Tuples with different keys can be sent to different operator instances, allowing parallel processing.

Stream operations often need to store state. For example, a windowing operator may need to retain all tuples that it has seen in a particular window, as long as the window is active. Or, it may need to store aggregates computed at some resolution (say per minute) to later compute coarser resolution aggregates (say per hour). There are many other reasons for operators to store state. User-defined operators often define state internal to the operator (local variables), which needs to be stored.

Such state may be stored locally, at each node where a copy of the operator is executed. Alternatively, it may be stored centrally in a parallel data-storage system. The store-locally alternative has a lower cost but a higher risk of losing state information on failure, as compared to storing state in a parallel data-storage system. This aspect is discussed further in Section 22.8.3.

22.8.3 Fault Tolerance with Streaming Data

Fault tolerance when querying stored data can be achieved by reexecuting the query, or parts of the query, as we have seen in Section 22.5.4. However, such an approach to fault tolerance does not work well in a streaming setting for multiple reasons. First, many streaming data applications are latency sensitive, and delays in delivering results due to restarts are not desirable. Second, streaming systems provide a continuous stream of outputs. In the event of a failure, reexecuting the entire system or parts of it could potentially lead to duplicate copies of output tuples, which is not acceptable for many applications.

Thus, streaming data systems need to provide guarantees about delivery of output tuples, which can be one of: at-least once, at-most once, and exactly-once. The **at-least-once** semantics guarantees that each tuple is output at least once, but allows duplicate delivery during recovery from failures. The **at-most-once** semantics guarantees that each tuple is delivered at most once, without duplicates, but some tuples may be lost in the event of a failure. The **exactly-once** semantics guarantees that each tuple will be delivered exactly once, regardless of failures. This is the model that most applications require, although some applications may not care about duplicates and may accept at-least-once semantics.

Telegram: @uni_k

www.konkur.in

To ensure such semantics, streaming systems must track what tuples have been processed at each operator and what tuples have been output. Duplicates can be detected by comparing against tuples output earlier and removed. (This can be done only if the system guarantees the absence of duplicates during normal processing, since otherwise the semantics of the streaming query may be affected by removal of genuine duplicates.)

One way to implement fault tolerance is to support it in the subsystem that routes tuples between operators. For example, in Kafka, tuples are published to topics, and each topic-partition can be stored in two or more nodes so that even if one of the nodes fails, the other one is available. Further, the tuples are stored on disk in each of the nodes so that they are not lost on power failure or system restart. Thus, the streaming data system can use this underlying fault tolerance and high availability mechanism to implement fault-tolerance and high availability at a higher level of the system.

In such a system, if an operator was executing on a failed node, it can be restarted on another node. The system must also (at least periodically) record up to what point each input stream had been consumed by the operator. The operator must be restarted and each input stream replayed from a point such that the operator can correctly output all tuples that it had not already output before the failure. This is relatively easy for operators without any state; operators without any state need to do extra work to restore the state that existed before failure. For example, a window operator needs to start from a point in the stream corresponding to the start of a window and replay tuples from that point.

If the window is very large, restarting from a very old point in the stream would be very inefficient. Instead, the operator may checkpoint its state periodically, along with points in the input stream up to which processing has been done. In the event of a failure, the latest checkpoint may be restored, and only input stream tuples that were processed since the last checkpoint need to be replayed.

The same approach can be for other operators that have state information; the state can be checkpointed periodically, and replay starts from the last checkpoint. The checkpointed state may be stored locally; however, this means that until the node recovers, stream processing cannot proceed. As an alternative, the state may be stored in a distributed file system or in a parallel data-storage system. Such systems replicate data to ensure high availability even in the event of failures. Thus, if a node has failed, its functions can be restarted on a different node, starting with the last checkpoint, and replaying the stream contents.

If the underlying system does not implement fault tolerance, operators can implement their own fault-tolerance mechanisms to avoid tuple loss. For example, each operator may store all tuples that it has output; a tuple can be discarded only after the operator knows that no consumer will need the tuple, even in the event of a failure.

Further, streaming systems must often guarantee low latency, even in the event of failures. To do so, some streaming systems have replicas of each operator, running concurrently. If one replica fails, the output can be fetched from the other replica. The system must make sure that duplicate tuples from the replicas are not output to

Telegram: @uni_k

www.konkur.in

consumers. In such systems, one copy of an operator is treated as a primary copy, and the other copy as a hot-spare replica (recall that we discussed hot-spares in Section 19.7).

What we have described above is a high-level view of how streaming data systems implement fault tolerance. References to more information on how to implement fault tolerance in streaming systems may be found in the Further Reading section at the end of the chapter.

22.9 Distributed Query Processing

The need for distributed query processing originally arose when organizations needed to execute queries across multiple databases that were often geographically distributed. However, today the same need arises because organizations have data stored in multiple different databases and data storage systems, and they need to execute queries that access multiples of these databases and data storage systems.

22.9.1 Data Integration from Multiple Data Sources

Different parts of an enterprise may use different databases, either because of a legacy of how they were automated, or because of mergers of companies. Migrating an entire organization to a common system may be an expensive and time-consuming operation. An alternative is to keep data in individual databases, but to provide users with a logical view of integrated data. The local database systems may employ different logical models, different data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms. Some of the sources of data may not be full-fledged database systems but may instead be data storage systems, or even just files in a file system. Yet another possibility is that the data source may be on the cloud and accessed as a web service. Queries may need access to data stored across multiple databases and data sources.

Manipulation of information located in multiple databases and other data sources requires an additional software layer on top of existing database systems. This layer creates the illusion of logical database integration without requiring physical database integration and is sometimes called a **federated database system**.

Database integration can be done in several different ways:

- The **federated database** approach creates a common schema, called a **global schema**, for data from all the databases/data sources; each database has its own **local schema**. The task of creating a unified global schema from multiple local schemas is referred to as **schema integration**.

  Users can issue queries against the global schema. A query on a global schema must be translated into queries on the local schemas at each of the sites where the query has to be executed. The query results have to be translated back into the global schema and combined to get the final result.

Telegram: @uni_k

www.konkur.in

In general, updates to the common schema also need to be mapped to updates to the individual databases; systems that support a common schema and queries, but not updates, against the schema are sometimes referred to as **mediator** systems.

- The **data virtualization** approach allows applications to access data from multiple databases/data sources, but it does not try to enforce a common schema. Users have to be aware of the different schemas used in different databases, but they do not need to worry about which data are stored on which database system, or about how to combine information from multiple databases.
- The **external data** approach allows database administrators to provide schema information about data that are stored in other databases, along with other information, such as connection and authorization information needed to access the data. Data stored in external sources that can be accessed from a database are referred to as **external data**. **Foreign tables** are views defined in a database whose actual data are stored in an external data source. Such tables can be read as well as updated, depending on what operations the external data source supports. Updates on foreign tables, if supported, must be translated into updates on the external data source.

  Unlike the earlier-mentioned approaches, the goal here is not to create a full-fledged distributed database, but merely to facilitate access to data from other data sources. The SQL Management of External Data (SQL MED) component of the SQL standard defines standards for accessing external data sources from a database.

If a data source is a database that supports SQL, its data can be easily accessed using ODBC or JDBC connections. Data in parallel data storage systems that do not support SQL, such as HBase, can be accessed using the API methods that they provide.

A **wrapper** provides a view of data stored at a data source, in a desired schema. For example, if the system has a global schema, and the local database schema is different from the global schema, a wrapper can provide a view of the data in the global schema. Wrappers can even be used to provide a relational view of nonrelational data sources, such as web services, flat files (e.g., web logs), and directory systems.

Wrappers can also translate queries on the global schema into queries on the local schema, and translate results back into the global schema. Wrappers may be provided by individual sites, they may be written as part of the federated database system. Many relational databases today support wrappers that provide a relational view of data stored in file systems; such wrappers are specific to the type of data stored in the files.

If the goal of data integration is solely to run decision support queries, *data warehouses*, which we saw in Section 11.2, are a widely used alternative to database integration. Data warehouses import data from multiple sources into a single (possibly parallel) system, with a centralized schema. Data are typically imported periodically,

Telegram: @uni_k

www.konkur.in

for example, once in a day or once in a few hours, although continuous import is also increasingly used. Raw data imported from the data sources are typically processed and cleaned before being stored in the data warehouse.

However, with the scale at which data are generated by some applications, creating and maintaining such a warehouse can be expensive. Furthermore, queries cannot access the most recent data, since there is a delay between updates on the source database and import of the updates to the data warehouse. On the other hand, query processing can be done more efficiently in a data warehouse; further, queries at a data warehouse do not affect the performance of other queries and transactions at the data source. Whether to use a data warehouse architecture, or to directly access data from the data sources in response to individual queries, is a decision that each enterprise has to make based on its needs.

The term **data lake** is used to refer to an architecture where data are stored in multiple data storage systems and in different formats, including in file systems, but can be queried from a single system. Data lakes are viewed as an alternative to data warehouses, since they do not require up-front effort to preprocess data, although they do require more effort when creating queries.

22.9.2 Schema and Data Integration

The first task in providing a unified view of data lies in creating a unified conceptual schema, a task that is referred to as **schema integration**. Each local system provides its own conceptual schema. The database system must integrate these separate schemas into one common schema. Schema integration is a complicated task, mainly because of the semantic heterogeneity. The same attribute names may appear in different local databases but with different meanings.

Schema integration requires the creation of a **global schema**, which provides a unified view of data in different databases. Schema integration also requires a way to define how data are mapped from the local schema representation at each database, to the global schema. This step can be done by defining views at each site which, transform data from the local schema to the global schema. Data in the global schema is then treated as the union of the global views at the individual site. This approach is called the **global-as-view** (**GAV**) approach.

Consider an example with two sites which store student information in two different ways:

- Site *s1* which uses the relation *student1*(*ID*, *name*, *dept_name*), and the relation *studentCreds*(*ID*, *tot_cred*).
- Site *s2* which uses the relation *student2*(*ID*, *name*, *tot_cred*), and the relation *studentDept*(*ID*, *dept_name*).

Let the global schema chosen be *student*(*ID*, *name*, *dept_name*, *tot_cred*).

Telegram: @uni_k

www.konkur.in

Then, the global schema view at site *s1* would be defined as the view:

```
create view student_s1(ID, name, dept_name, tot_cred) as
select ID, name, dept_name, tot_cred
from student1, studentCreds
where student1.ID= studentCreds.ID;
```

While the global schema view at site *s2* would be defined as the view:

```
create view student_s2(ID, name, dept_name, tot_cred) as
select ID, name, dept_name, tot_cred
from student2, studentDept
where student2.ID= studentDept.ID;
```

Finally, the global schema *student* would be defined as the union of *student_s1* and *student_s2*.

Note that with the above view definition, a query on the global schema relation *student* can be easily translated into queries on the local schema relations at the sites *s1* and *s2*. It is harder to translate updates on the global schema into updates on the local schema, since there may not be a unique way to do so as discussed in Section 4.2.

There are more complex mapping schemes that are designed to deal with duplication of information across sites and to allow translation of updates on the global schema into updates on the local schema. The **local-as-view** (**LAV**) approach, which defines local data in each site as a view on a conceptual unified global relation, is one such approach.

Consider for example a situation where the *student* relation is partitioned between two sites based on the *dept_name* attribute, with all students in the "Comp. Sci." department at site *s3* and all students in other departments in site *s4*. This can be specified using the local-as-view approach by defining the relations *student_s3* and *student_s4*, which are actually stored at the sites *s3* and *s4*, as equivalent to views defined on the global relation *student*.

```
create view student_s3 as
select ∗
from student
where student.dept_name = 'Comp. Sci.';
```

and

```
create view student_s4 as
select ∗
from student
where student.dept_name ! = 'Comp. Sci.';
```

Telegram: @uni_k

www.konkur.in

With this extra information, the query optimizer can figure out that a query that seeks to retrieve students in the Comp. Sci. department need only be executed at site *s3* and need not be executed at site *s4*. More information on schema integration may be found in the bibliographic notes for this chapter, available online.

The second task in providing a unified view of data from multiple sources lies in dealing with differences in data types and values. For example, the data types used in one system may not be supported by other systems, and translation between types may not be simple. Even for identical data types, problems may arise from the physical representation of data: One system may use 8-bit ASCII, while another may use 16-bit Unicode; floating-point representations may differ; integers may be represented in *big-endian* or *little-endian* form. At the semantic level, an integer value for length may be inches in one system and millimeters in another; when integrating the data, a single representation must be used, and values converted to the chosen representation. Mapping between types can be done as part of the view definitions that translate data between the local schemas and the global schema.

A deeper problem is that the same conceptual entity may have different names in different systems. For example, a system based in the United States may refer to the city "Cologne," whereas one in Germany refers to it as "Köln." One approach to deal with this problem is to have a globally unique naming system, and map values to the unique names as part of the view definitions used for schema mappings. For example, the International Standards Organization has a unique code for country names, and for states/provinces within the countries. The GeoNames database (www.geonames.org) provides unique names for several million locations such as cities, geographical features, roads, buildings, and so forth.

When such standard naming systems are not available, some systems allow the specification of name equivalences; for example, such a system could allow a user to say that "Cologne" is the same as "Köln". This approach is used in the **Linked Data** project, which supports the integration of a very large number of databases that use the RDF representation of data (the RDF representation is described in Section 8.1.4). However, querying is more complicated in such a scenario.

In our description of the view definitions above, we assumed that data are stored in local databases, and the view definitions are used to provide a global view of the data, without actually materializing the data in the global schema. However, such views can also be used to materialize the data in the global schema, which can then be stored in a data warehouse. In the latter case, updates on underlying data must be propagated to the data warehouse.

22.9.3 Query Processing Across Multiple Data Sources

A naive way to execute a query that accesses data from multiple data sources is to fetch all required data to one database, which then executes the query. But suppose, for example, that the query has a selection condition that is satisfied by only one or a few records out of a large relation. If the data source allows the selection to be performed at

Telegram: @uni_k

www.konkur.in

the data source, it makes no sense to retrieve the entire relation; instead, the selection operation should be performed at the data source, while other operations, if any, may be performed at the database where the query was issued.

In general, different data sources may support different query capabilities. For example, if the source is a data storage system, it may support selections on key attributes only. Web data sources may restrict which fields selections are allowed on and may additionally require that selections be present on certain fields. On the other hand, if the source is a database that supports SQL, operations such as join or aggregation could be performed at the source and only the result brought over to the database that issues the query. In general, queries may have to be broken up and performed partly at the data source and partly at the site issuing the query.

The cost of processing a query that accesses multiple data sources depends on the local execution costs, as well as on the data transfer cost. If the network is a low bandwidth wide-area network, particular attention must be paid to minimizing data transfer.

In this section, we study issues in distributed query processing and optimization.

22.9.3.1 Join Locations and Join Ordering

Consider the following relational-algebra expression:

$$r_1 \bowtie r_2 \bowtie r_3$$

Assume that r_1 is stored at site S_1, r_2 at S_2, and r_3 at S_3. Let S_I denote the site at which the query was issued. The system needs to produce the result at site S_I. Among the possible strategies for processing this query are these:

- Ship copies of all three relations to site S_I. Using the techniques of Chapter 16, choose a strategy for processing the entire query locally at site S_I.
- Ship a copy of the r_1 relation to site S_2, and compute $temp_1 = r_1 \bowtie r_2$ at S_2. Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie r_3$ at S_3. Ship the result $temp_2$ to S_I.
- Devise strategies similar to the previous one, with the roles of S_1, S_2, S_3 exchanged.

There are several other possible strategies.

No one strategy is always the best one. Among the factors that must be considered are the volume of data being shipped, the cost of transmitting a block of data between a pair of sites, and the relative speed of processing at each site. Consider the first strategy. Suppose indices present at S_2 and S_3 are useful for computing the join. If we ship all three relations to S_I, we would need to either re-create these indices at S_I or use a different, possibly more expensive, join strategy. Re-creation of indices entails extra processing overhead and extra disk accesses. There are many variants of the second strategy, which process joins in different orders.

Telegram: @uni_k

www.konkur.in

The cost of each of the strategies depends on the sizes of the intermediate results, the network transmission costs, and the costs of processing at each node. The query optimizer needs to choose the best strategy, based on cost estimates.

22.9.3.2 Semijoin Strategy

Suppose that we wish to evaluate the expression $r_1 \bowtie r_2$, where r_1 and r_2 are stored at sites S_1 and S_2, respectively. Let the schemas of r_1 and r_2 be R_1 and R_2. Suppose that we wish to obtain the result at S_1. If there are many tuples of r_2 that do not join with any tuple of r_1, then shipping r_2 to S_1 entails shipping tuples that fail to contribute to the result. We want to remove such tuples before shipping data to S_1, particularly if network costs are high.

A possible strategy to accomplish all this is:

1. Compute $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$ at S_1.
2. Ship $temp_1$ from S_1 to S_2.
3. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ at S_2.
4. Ship $temp_2$ from S_2 to S_1.
5. Compute $r_1 \bowtie temp_2$ at S_1. The resulting relation is the same as $r_1 \bowtie r_2$.

Before considering the efficiency of this strategy, let us verify that the strategy computes the correct answer. In step 3, $temp_2$ has the result of $r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$. In step 5, we compute:

$$r_1 \bowtie r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$$

Since join is associative and commutative, we can rewrite this expression as:

$$(r_1 \bowtie \Pi_{R_1 \cap R_2}(r_1)) \bowtie r_2$$

Since $r_1 \bowtie \Pi_{(R_1 \cap R_2)}(r_1) = r_1$, the expression is, indeed, equal to $r_1 \bowtie r_2$, the expression we are trying to evaluate.

This strategy is called a **semijoin strategy**, after the semijoin operator of the relational algebra, denoted $\ltimes$, which we saw in Section 16.4.4. The *natural semijoin* of r_1 with r_2, denoted $r_1 \ltimes r_2$, is defined as:

$$r_1 \ltimes r_2 \stackrel{\text{def}}{=} \Pi_{R_1}(r_1 \bowtie r_2)$$

Thus, $r_1 \ltimes r_2$ selects those tuples of relation r_1 that contributed to $r_1 \bowtie r_2$. In step 3, $temp_2 = r_2 \ltimes r_1$. The semijoin operation is easily extended to theta-joins. The *theta semijoin* of r_1 with r_2, denoted $r_1 \ltimes_\theta r_2$, is defined as:

$$r_1 \ltimes_\theta r_2 \stackrel{\text{def}}{=} \Pi_{R_1}(r_1 \bowtie_\theta r_2)$$

Telegram: @uni_k

www.konkur.in

For joins of several relations, the semijoin strategy can be extended to a series of semijoin steps. It is the job of the query optimizer to choose the best strategy based on cost estimates.

This strategy is particularly advantageous when relatively few tuples of r_2 contribute to the join. This situation is likely to occur if r_1 is the result of a relational-algebra expression involving selection. In such a case, $temp_2$, that is, $r_2 \ltimes r_1$, may have significantly fewer tuples than r_2. The cost savings of the strategy result from having to ship only $r_2 \ltimes r_1$, rather than all of r_2, to S_1.

Some additional cost is incurred in shipping $temp_1$, that is $\Pi_{R_1 \cap R_2}(r_1)$ to S_2. If a sufficiently small fraction of tuples in r_2 contribute to the join, the overhead of shipping $temp_1$ will be dominated by the savings of shipping only a fraction of the tuples in r_2. The overhead of sending $temp_1$ tuples from s_1 to s_2 can be reduced as follows. For the purpose of optimization of join processing, the semijoin operation can be implemented in a manner that overapproximates the true semijoin result. That is, the result should contain all the tuples in the actual semijoin result, but it may contain a few extra tuples. The extra tuples will get eliminated later by the join operation.

An efficient overapproximation of the semijoin result can be computed by using a probabilistic data structure called a **Bloom filter**, which uses bitmaps. Bloom filters are described in more detail in Section 24.1. To implement $r_2 \ltimes r_1$, a Bloom filter with a bitmap b of size m, initialized with all bits set to 0 is used. The join attributes of each tuple of r_1 are hashed to a value in the range $0 \ldots (m - 1)$, and the corresponding bit of b is set to 1. The bitmap b, which is much smaller than the relation r_1, can now be sent to the site containing r_2. There, the same hash function is computed on the join attributes of each tuple of r_2. If the corresponding bit is set to 1 in b, that r_2 tuple is accepted (added to the *result* relation), and otherwise it is rejected.

Note that it is possible for different join attribute values, say v_1 and v_2 to have the same hash value; even if r_1 has a tuple with value v_1, but does not have any tuple with value v_2, the result of the above procedure may include r_2 tuples whose join attribute value is v_2. Such a situation is referred to as a **false positive**. However, if v_1 is present in r_1, the technique will never reject a tuple in r_2 that has join attribute value v_1, which is important for correctness.

The *result* relation computed above, which is a superset of or equal to $r_2 \ltimes r_1$, is sent to site s_1. The join $r_1 \bowtie$ *result* is then computed at site s_1 to get the required join result. False positives may result in extra tuples in *result* that are not present in $r_2 \ltimes r_1$, but such tuples would be eliminated by the join.

To keep the probability of false positives low, the number of bits in the Bloom filter is usually set to a few times the estimated number of distinct join attribute values. Further, it is possible to use k independent hash functions, for some $k > 1$, to identify k bit positions for a given value, and set all of them to 1 when creating the bitmap. When querying it with a given value, the same k hash functions are used to identify k bit locations, and the value is determined to be absent if even one of the k bits has a 0 value. For example, if the bitmap has $10n$ bits, where n is the number of distinct join

Telegram: @uni_k

www.konkur.in

attribute values, and $k = 7$ hash functions are used, the false positive rate would be about 1%.

22.9.3.3 Distributed Query Optimization

Several extensions need to be made to existing query optimization techniques in order to optimize distributed query plans.

The first extension is to record the location of data as a *physical property* of the data; recall that optimizers already deal with other physical properties such as the sort order of results. Just as the sort operation is used to create different sort orders, an exchange operation is used to transfer data between different sites.

The second extension is to track where an operator is executed; optimizers already consider different algorithms, such as hash join or merge join, for a given logical operator, in this case, the join operator. The optimizer is extended to additionally consider alternative sites for execution of each algorithm. Note that to execute an operator at a given site, its inputs must satisfy the physical property of being located at that site.

The third extension is to consider semijoin operations to reduce data transfer costs. Semijoin operations can be introduced as logical transformation rules; however, if done naively, the search space increases greatly, making this approach infeasible. Optimization cost can be reduced by restricting, as a heuristic, semijoins to be applied only on database tables, and never on intermediate join results.

A fourth extension is to use schema information to restrict the set of nodes at which a query needs to be executed. Recall from Section 22.9.2 that the local-as-view approach can be used to specify that a relation is partitioned in a particular way. In the example we saw there, site *s3* contains all student tuples with *dept_name* being Comp. Sci., while *s4* contains all the other student tuples. Suppose a query has a selection "*dept_name*='Comp. Sci.'" on *student*; then, the optimizer should recognize that there is no need to involve site *s4* when executing this query.

As another example, if the student data at site *s5* is a replica of the data at site *s3*, then the optimizer can choose to execute the query at either of the sites, depending on which is cheaper; there is no need to execute the query at both sites.

22.9.4 Distributed Directory Systems

A directory is a listing of information about some class of objects such as persons. Directories can be used to find information about a specific object, or in the reverse direction to find objects that meet a certain requirement. Several **directory access protocols** have been developed to provide a standardized way of accessing data in a directory.

A very widely used distributed directory system is the internet **Domain Name Service** (**DNS**) system, which provides a standardized way to map domain names (such as db-book.com or www.cs.yale.edu, to the IP addresses of the machines. (Although users see only the domain names, the underlying network routes messages based on IP addresses, and hence a way to convert domain names to IP addresses is critical for

Telegram: @uni_k

www.konkur.in

the functioning of the internet.) The **Lightweight Directory Access Protocol** (**LDAP**) is another very widely used protocol designed for storing organizational data.

Data stored in directories can be represented in the relational model, stored in a relational database, and accessed through standard protocols such as JDBC or ODBC. The question then is, why come up with a specialized protocol for accessing directory information? There are several reasons.

- First, directory access protocols are simplified protocols that cater to a limited type of access to data. They evolved in parallel with the database access protocol standards.
- Second, directory systems were designed to support a hierarchical naming system for objects, similar to file system directory names. Such a naming system is important in many applications. For example, all computers whose names end in yale.edu belong to Yale, while those whose names end in iitb.ac.in belong to IIT Bombay. Within the yale.edu domain, there are subdomains such as cs.yale.edu, which corresponds to the CS department in Yale, and math.yale.edu which corresponds to the Math department at Yale.
- Third, and most important from a distributed systems perspective, the data in a distributed directory system are stored and controlled in a distributed, hierarchical, manner.

  For example, a DNS server at Yale would store information about names of computers at Yale, along with associated information such as their IP addresses. Similarly, DNS servers at Lehigh and IIT Bombay would store information about computers in their respective domains. The DNS servers store information in a hierarchical fashion; for example, the information provided by the Yale DNS server may be stored in a distributed fashion across subdomains at Yale, such as the CS and Math DNS servers.

  Distributed directory systems automatically forward queries submitted at a site to the site where the required information is actually stored, to give a unified view of data to users and applications.

  Further, distributed directory implementations typically support replication to ensure the availability of data even if some nodes have failed.

Another example of usage of directory systems is for organization data. Such systems store information about employees, such as the employee identifier, name, email, organization unit (such as department), room number, phone number, and (encrypted) password of each employee. The schema of such organizational data are standardized as part of the Lightweight Directory Access Protocol (LDAP). Directory systems based on the LDAP protocol are widely used to authenticate users, using the encrypted passwords stored for each user. (More information about the LDAP data representation may be found in Section 25.5.)

Telegram: @uni_k

www.konkur.in

Although distributed data storage across organizational units was important at one time, such directory systems are often centralized these days. In fact, several directory implementations use relational databases to store data, instead of creating special-purpose storage systems. However, the fact that the data representation and protocol to access the data are standardized has meant that these protocols continue to be widely used.

22.10 Summary

- Current generation parallel systems are typically based on a hybrid architecture, where each computer has multiple cores with a shared memory, and there are multiple computers organized in a shared-nothing fashion.
- Parallel processing in a database system can be exploited in two distinct ways.
  - Interquery parallelism—the execution of multiple queries in parallel with each other, across multiple nodes.
  - Intraquery parallelism—the processing of different parts of the execution of a single query, in parallel across multiple nodes.
- Interquery parallelism is essential for transaction-processing systems, while intraquery parallelism is essential for speeding up long-running queries.
- Execution of a single query involves execution of multiple relational operations. The key to exploiting large-scale parallelism is to process each operation in parallel, across multiple nodes (referred to as intraoperation parallelism).
- The operations that are the most amenable to parallelism are: sort, selection, duplicate elimination, projection, and aggregation.
- Range-partitioning sort works in two steps: first range-partitioning the relation, then sorting each partition separately.
- Parallel external sort-merge works in two steps: first, each node N_i sorts the data available at node N_i, then the system merges the sorted runs on each node to get the final sorted output.
- Parallel join algorithms divide the tuples of the input relations over several nodes. Each node then computes part of the join locally. Then the system collects the results from each node to produce the final result.
- Skew is a major problem, especially with increasing degrees of parallelism. Balanced partitioning vectors, using histograms, and virtual node partitioning are among the techniques used to reduce skew.

Telegram: @uni_k

www.konkur.in

- The MapReduce paradigm is designed to ease the writing of parallel data processing programs using imperative programming languages that may not be expressible using SQL. Fault-tolerant implementations of the MapReduce paradigm are important for a variety of very large scale data processing tasks. Extensions of the MapReduce model based on algebraic operations is increasingly important. The Hive SQL system uses a MapReduce system as its underlying execution engine, compiling SQL queries to MapReduce code.
- There are two forms of interoperation parallelism: pipelined parallelism and independent parallelism. Pipelined parallelism is usually implemented using a push model, with buffers between operations.
- The exchange operation repartitions data in a specified way. Parallel query plans can be created in such a way that data interchange between nodes is done only by the exchange operator, while all other operations work on local data, just as they would in a centralized database system.
- In the event of failure, parallel query execution plans could be restarted. However, in very large systems where there is a significant chance of failure during the execution of a query, fault tolerance techniques are important to ensure that queries complete execution without restarting, despite failures.
- Parallel algorithms designed for shared-nothing architectures can be used in shared-memory architectures. Each processor can be treated as having its own partition of memory, and we can ignore the fact that the processors have a common shared memory. However, execution can be optimized significantly by exploiting the fast access to shared memory from any of the processors.
- Query optimization for parallel execution can be done using the traditional resource consumption cost model, or using the response time cost model. Partitioning of tables must be taken into account when choosing a plan, to minimize data exchange which is often a significant factor in query execution cost. Materialized views can be important in parallel environments since they can significantly reduce query execution cost.
- There are many streaming data applications today that require high performance processing, which can only be achieved by parallel processing of streaming data. Incoming tuples and results of operations need to be routed to other operations. The publish-subscribe model, implemented for example in Apache Kafka, has proven quite useful for such routing. Fault-tolerant processing of streaming data with exactly-once semantics for processing tuples is important in many applications. Persistence provided by publish-subscribe systems helps in this regard.
- Integration of schema and data from multiple databases is needed for many data processing tasks. The external data approach allows external data to be queries in a database as if it is locally resident. The global-as-view and local-as-view architec-

Telegram: @uni_k

www.konkur.in

tures allows rewriting of queries from a global schema to individual local schemas. The semijoin strategy using Bloom filters can be useful to reduce data movement for joins in a distributed database. Distributed directory systems are a type of distributed database designed for distributed storage and querying of directories.

Review Terms

- Interquery parallelism
- Intraquery parallelism
  - Intraoperation parallelism
  - Interoperation parallelism
- Range-partitioning sort
- Data parallelism
- Parallel external sort-merge
- Partitioned join
- Partitioned parallel hash join
- Partitioned parallel merge join
- Partitioned parallel nested-loop join
- Partitioned parallel indexed nested-loops join
- Asymmetric fragment-and-replicate join
- Broadcast join
- Fragment-and-replicate join
- Symmetric fragment-and-replicate join
- Join skew avoidance
- Dynamic handling of join skew
- Work stealing
- Parallel selection
- Parallel duplicate elimination
- Parallel projection
- Parallel aggregation
- Partial aggregation
- Intermediate key
- Pipelined parallelism
- Independent parallelism
- Exchange operator
- Unpartitioned
- Random merge
- Ordered merge
- Parallel query execution plan
- Query processing in shared memory
- Thread
- Single Instruction Multiple Data (SIMD)
- Response-time cost model
- Parallel view maintenance
- Streaming data
- Lambda architecture
- Routing of streams
- Publish-subscribe
- Topic-partition
- At-least-once semantics
- At-most-once semantics
- Exactly-once semantics
- Federated database system
- Global schema
- Local schema
- Schema integration
- Mediator
- Data virtualization
- External data
- Foreign tables

Telegram: @uni_k

www.konkur.in

- Data lake
- Global-as-view (GAV)
- Local-as-view (LAV)
- Linked Data
- Semijoin strategy
- Semijoin
- Theta semijoin
- Bloom filter
- False positive
- Directory access protocols
- Domain Name Service (DNS)
- Lightweight Directory Access Protocol (LDAP)

Practice Exercises

22.1 What form of parallelism (interquery, interoperation, or intraoperation) is likely to be the most important for each of the following tasks?

a. Increasing the throughput of a system with many small queries

b. Increasing the throughput of a system with a few large queries when the number of disks and processors is large

22.2 Describe how partial aggregation can be implemented for the **count** and **avg** aggregate functions to reduce data transfer.

22.3 With pipelined parallelism, it is often a good idea to perform several operations in a pipeline on a single processor, even when many processors are available.

a. Explain why.

b. Would the arguments you advanced in part *a* hold if the machine has a shared-memory architecture? Explain why or why not.

c. Would the arguments in part *a* hold with independent parallelism? (That is, are there cases where, even if the operations are not pipelined and there are many processors available, it is still a good idea to perform several operations on the same processor?)

22.4 Consider join processing using symmetric fragment and replicate with range partitioning. How can you optimize the evaluation if the join condition is of the form $|\ r.A - s.B\ | \leq k$, where k is a small constant? Here, $|\ x\ |$ denotes the absolute value of x. A join with such a join condition is called a **band join**.

22.5 Suppose relation r is stored partitioned and indexed on A, and s is stored partitioned and indexed on B. Consider the query:

$$_{r.C}\gamma_{\text{count}(s.D)}(\ (\sigma_{A>5}(r)) \bowtie_{r.B=s.B} s\ )$$

a. Give a parallel query plan using the exchange operator, for computing the subtree of the query involving only the select and join operators.

Telegram: @uni_k

www.konkur.in

b. Now extend the above to compute the aggregate. Make sure to use pre-aggregation to minimize the data transfer.

c. Skew during aggregation is a serious problem. Explain how pre-aggregation as above can also significantly reduce the effect of skew during aggregation.

22.6 Suppose relation r is stored partitioned and indexed on A, and s is stored partitioned and indexed on B. Consider the join $r \bowtie_{r.B=s.B} s$. Suppose s is relatively small, but not small enough to make asymmetric fragment-and-replicate join the best choice, and r is large, with most r tuples not matching any s tuple. A hash-join can be performed but with a semijoin filter used to reduce the data transfer. Explain how semijoin filtering using Bloom filters would work in this parallel join setting.

22.7 Suppose you want to compute $r \mathbin{⟕}_{r.A=s.A} s$.

a. Suppose s is a small relation, while r is stored partitioned on $r.B$. Give an efficient parallel algorithm for computing the left outer join.

b. Now suppose that r is a small relation, and s is a large relation, stored partitioned on attribute $s.B$. Give an efficient parallel algorithm for computing the above left outer join.

22.8 Suppose you want to compute ${}_{A,B}\gamma_{sum(C)}$ on a relation s which is stored partitioned on $s.B$. Explain how you would do it efficiently, minimizing/avoiding repartitioning, if the number of distinct $s.B$ values is large, and the distribution of number of tuples with each $s.B$ value is relatively uniform.

22.9 MapReduce implementations provide fault tolerance, where you can reexecute only failed mappers or reducers. By default, a partitioned parallel join execution would have to be rerun completely in case of even one node failure. It is possible to modify a parallel partitioned join execution to add fault tolerance in a manner similar to MapReduce, so failure of a node does not require full reexecution of the query, but only actions related to that node. Explain what needs to be done at the time of partitioning at the sending node and receiving node to do this.

22.10 If a parallel data-store is used to store two relations r and s and we need to join r and s, it may be useful to maintain the join as a materialized view. What are the benefits and overheads in terms of overall throughput, use of space, and response time to user queries?

22.11 Explain how each of the following join algorithms can be implemented using the MapReduce framework:

a. Broadcast join (also known as asymmetric fragment-and-replicate join).

Telegram: @uni_k

www.konkur.in

b. Indexed nested loop join, where the inner relation is stored in a parallel data-store.

c. Partitioned join.

Exercises

22.12 Can partitioned join be used for $r \bowtie_{r.A<s.A \wedge r.B=s.B} s$? Explain your answer.

22.13 Describe a good way to parallelize each of the following:

a. The difference operation

b. Aggregation by the **count** operation

c. Aggregation by the **count distinct** operation

d. Aggregation by the **avg** operation

e. Left outer join, if the join condition involves only equality

f. Left outer join, if the join condition involves comparisons other than equality

g. Full outer join, if the join condition involves comparisons other than equality

22.14 Suppose you wish to handle a workload consisting of a large number of small transactions by using shared-nothing parallelism.

a. Is intraquery parallelism required in such a situation? If not, why, and what form of parallelism is appropriate?

b. What form of skew would be of significance with such a workload?

c. Suppose most transactions accessed one *account* record, which includes an *account_type* attribute, and an associated *account_type_master* record, which provides information about the account type. How would you partition and/or replicate data to speed up transactions? You may assume that the *account_type_master* relation is rarely updated.

22.15 What is the motivation for work-stealing with virtual nodes in a shared-memory setting? Why might work-stealing not be as efficient in a shared-nothing setting?

22.16 The attribute on which a relation is partitioned can have a significant impact on the cost of a query.

a. Given a workload of SQL queries on a single relation, what attributes would be candidates for partitioning?

Telegram: @uni_k

www.konkur.in

b. How would you choose between the alternative partitioning techniques, based on the workload?

c. Is it possible to partition a relation on more than one attribute? Explain your answer.

22.17 Consider system that is processing a stream of tuples for a relation r with attributes (A, B, C, *timestamp*) Suppose the goal of a parallel stream processing system is to compute the number of tuples for each A value in each 5 minute window (based on the timestamp of the tuple). What would be the topic and the topic partitions? Explain why.

Tools

A wide variety of open-source tools are available, in addition to some commercial tools, for parallel query processing. A number of these tools are also available on hosted cloud platforms.

Teradata was one of the first commercial parallel database systems, and it continues to have a large market share. The Red Brick Warehouse was another early parallel database system; Red Brick was acquired by Informix, which was itself acquired by IBM. Other commercial parallel database systems include Teradata Aster Data, IBM Netezza, and Pivotal Greenplum. IBM Netezza, Pivotal Greenplum, and Teradata Aster Data all use PostgreSQL as the underlying database, running independently on each node; each of these systems builds a layer on top, to partition data, and parallelize query processing across the nodes.

The open source Hadoop platform (hadoop.apache.org) includes the HDFS distributed file system and the Hadoop MapReduce platform. Systems that support SQL on a MapReduce platform include Apache Hive (hive.apache.org), which originated at Facebook, Apache Impala (impala.apache.org), which originated at Cloudera, and Apache HAWQ (hawq.apache.org), which originated at Pivotal. Apache Spark (spark.apache.org), which originated at the Univ. of California, Berkeley, and Apache Tez (tez.apache.org) are parallel execution frameworks that support a variety of operators beyond the basic map and reduce operators; and Hive SQL queries can be executed on both these platforms. Other parallel execution frameworks include Apache Pig (pig.apache.org), which originated at Yahoo!, the Asterix system (asterix.ics.uci.edu), which originated at the University of California, Irvine, the Apache Flink system (flink.apache.org), which originated as the Stratosphere project at the Technical University, Berlin, Humboldt University and the Hasso-Plattner Institute). Apache Spark and Apache Flink also support libraries for parallel machine learning.

These systems can access data stored in multiple different data formats, such as files in different formats in HDFS, or objects in a storage system such as HBase. Hadoop file formats were initially textual files, but today Hadoop implementations support several optimized file formats such as Sequence files (which allow binary data), Avro

Telegram: @uni_k

www.konkur.in

(which supports semi-structured schemas) and Parquet (which supports columnar data representation).

Apache Kafka (kafka.apache.org) is widely used for routing tuples in streaming data systems. Systems designed for query processing on streaming data include Apache Storm (storm.apache.org), Kafka Streams (kafka.apache.org/documentation/streams/) and Heron (apache.github.io/incubator-heron/), developed by Twitter. Apache Flink (flink.apache.org), Spark Streaming (spark.apache.org/streaming/), the streaming component of Apache Spark, and Apache Apex (apex.apache.org) support analytics on streaming data along with analytics on stored data.

Many of the above systems are also offered as cloud-based services, as part of the cloud services offered by Amazon AWS, Google Cloud, Microsoft Azure, and other similar platforms.

Further Reading

Early work on parallel database systems include GAMMA ([DeWitt (1990)]), XPRS ([Stonebraker et al. (1989)]) and Volcano ([Graefe (1990)]). [Graefe (1993)] presents an excellent survey of query processing, including parallel processing of queries. The exchange-operator model was advocated by [Graefe (1990)] and [Graefe and McKenna (1993)]. Skew handling in parallel joins is described by [DeWitt et al. (1992)].

[Ganguly et al. (1992)] describe query-optimization techniques based on the response-time cost model for parallel query execution, while [Zhou et al. (2010)] describe how to extend a query optimizer to account for partitioning properties and parallel plans. View maintenance in parallel data storage systems is described by [Agrawal et al. (2009)].

A fault-tolerant implementation of the MapReduce framework at Google, which lead to the widespread use of the MapReduce paradigm, is described by [Dean and Ghemawat (2010)]. [Kwon et al. (2013)] provide an overview of skew handling in the Hadoop MapReduce framework. [Herodotou and Babu (2013)] describe how to optimize a number of parameters for query execution in the MapReduce framework. An overview of the Apache Spark system is provided by [Zaharia et al. (2016)], while [Zaharia et al. (2012)] describe Resilient Distributed Datasets, a fault-tolerant abstraction which formed a basis for the Spark implementation. Extensions of the exchange operator to support fault-tolerance, are described by [Shah et al. (2004)], with a focus on fault-tolerant continuous queries. Fault-tolerant stream processing in the Google MillWheel system is described in [Akidau et al. (2013)].

The morsel-driven approach to parallel query evaluation in shared-memory systems with multi-core processors is described in [Leis et al. (2014)]. [Kersten et al. (2018)] provides a comparison of vectorwise query processing using optimizations such as SIMD instructions, with producer-driven pipelining.

Telegram: @uni_k

www.konkur.in

[Carbone et al. (2015)] describe stream and batch processing in Apache Flink. [Ozsu and Valduriez (2010)] provides textbook coverage of distributed database systems.

Bibliography

[Agrawal et al. (2009)] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for VLSD databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2009), pages 179–192.

[Akidau et al. (2013)] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-tolerant Stream Processing at Internet Scale", *Proceedings of the VLDB Endowment*, Volume 6, Number 11 (2013), pages 1033–1044.

[Carbone et al. (2015)] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine", *IEEE Data Eng. Bull.*, Volume 38, Number 4 (2015), pages 28–38.

[Dean and Ghemawat (2010)] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool", *Communications of the ACM*, Volume 53, Number 1 (2010), pages 72–77.

[DeWitt (1990)] D. DeWitt, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Number 1 (1990), pages 44–62.

[DeWitt et al. (1992)] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins", In *Proc. of the International Conf. on Very Large Databases* (1992), pages 27–40.

[Ganguly et al. (1992)] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992), pages 9–18.

[Graefe (1990)] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 102–111.

[Graefe (1993)] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, Volume 25, Number 2 (1993).

[Graefe and McKenna (1993)] G. Graefe and W. McKenna, "The Volcano Optimizer Generator", In *Proc. of the International Conf. on Data Engineering* (1993), pages 209–218.

[Herodotou and Babu (2013)] H. Herodotou and S. Babu, "A What-if Engine for Cost-based MapReduce Optimization", *IEEE Data Eng. Bull.*, Volume 36, Number 1 (2013), pages 5–14.

[Kersten et al. (2018)] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz, "Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask", *Proceedings of the VLDB Endowment*, Volume 11, Number 13 (2018), pages 2209–2222.

Telegram: @uni_k

www.konkur.in

[Kwon et al. (2013)] Y. Kwon, K. Ren, M. Balazinska, and B. Howe, "Managing Skew in Hadoop", *IEEE Data Eng. Bull.*, Volume 36, Number 1 (2013), pages 24–33.

[Leis et al. (2014)] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2014), pages 743–754.

[Ozsu and Valduriez (2010)] T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd edition, Prentice Hall (2010).

[Shah et al. (2004)] M. A. Shah, J. M. Hellerstein, and E. A. Brewer, "Highly-Available, Fault-Tolerant, Parallel Dataflows", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), pages 827–838.

[Stonebraker et al. (1989)] M. Stonebraker, P. Aoki, and M. Seltzer, "Parallelism in XPRS", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989).

[Zaharia et al. (2012)] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", In *Procs. USENIX Symposium on Networked Systems Design and Implementation, NSDI* (2012), pages 15–28.

[Zaharia et al. (2016)] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: a unified engine for big data processing", *Communications of the ACM*, Volume 59, Number 11 (2016), pages 56–65.

[Zhou et al. (2010)] J. Zhou, P. Larson, and R. Chaiken, "Incorporating partitioning and parallel plans into the SCOPE optimizer", In *Proc. of the International Conf. on Data Engineering* (2010), pages 1060–1071.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 23



Parallel and Distributed Transaction Processing

We studied transaction processing in centralized databases earlier, covering concurrency control in Chapter 18 and recovery in Chapter 19. In this chapter, we study how to carry out transaction processing in parallel and distributed databases. In addition to supporting concurrency control and recovery, transaction processing in parallel and distributed databases must also deal with issues due to replication of data, and of failures of some nodes.

Both parallel and distributed databases have multiple nodes, which can fail independently. The main difference between parallel and distributed databases from the view point of transaction processing is that the latency of remote access is much higher, and bandwidth lower, in a distributed database than in a parallel database where all nodes are in a single data center. Failures such as network partitioning and message delays are much less likely within a data center than across geographically distributed sites, but nevertheless they can occur; transaction processing must be done correctly even if they do occur.

Thus, most techniques for transaction processing are common to both parallel and distributed databases. In the few cases where there is a difference, we explicitly point out the difference. And as a result, in this chapter, whenever we say that a technique is applicable to distributed databases, it should be interpreted to mean that it is applicable to distributed databases as well as to parallel databases, unless we explicitly say otherwise.

In Section 23.1, we outline a model for transaction processing in a distributed database. In Section 23.2, we describe how to implement atomic transactions in a distributed database by using special commit protocols.

In Section 23.3 we describe how to extend traditional concurrency control techniques to distributed databases. Section 23.4 describes concurrency control techniques for the case where data items are replicated, while Section 23.5 describes further extensions including how multiversion concurrency control techniques can be extended to deal with distributed databases, and concurrency control can be implemented with

Telegram: @uni_k

www.konkur.in

heterogeneous distributed databases. Replication with weak degrees of consistency is discussed in Section 23.6.

Most techniques for dealing with distributed data require the use of coordinators to ensure consistent and efficient transaction processing. In Section 23.7 we discuss how coordinators can be chosen in a distributed fashion, robust to failures. Finally, Section 23.8 describes the distributed consensus problem, outlines solutions for the problem, and then discusses how these solutions can be used to implement fault-tolerant services by means of replication of a log.

23.1 Distributed Transactions

Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties (Section 17.1). There are two types of transaction that we need to consider. The **local transactions** are those that access and update data in only one local database; the **global transactions** are those that access and update data in several local databases. Ensuring the ACID properties of the local transactions can be done as described in Chapter 17, Chapter 18, and Chapter 19. However, for global transactions, this task is much more complicated, since several nodes may be participating in the execution of the transaction. The failure of one of these nodes, or the failure of a communication link connecting these nodes, may result in erroneous computations.

In this section, we study the system structure of a distributed database and its possible failure modes. In later sections, we discuss how to ensure ACID properties are satisfied in a distributed database, despite failures. We reemphasize that these failure modes occur with parallel databases as well, and the techniques we describe are equally applicable to parallel databases.

23.1.1 System Structure

We now consider a system structure with multiple nodes, each of which can fail independently of the others. We note that the nodes may be within a single data center, corresponding to a parallel database system, or geographically distributed, in a distributed database system. The system structure is similar in either case; the problems with respect to transaction isolation and atomicity are the same in both cases, as are the solutions.

We note that the system structure we consider here is not applicable to a shared-memory parallel database system whose components do not have independent modes of failures. In such systems either the whole system is up, or the whole system is down. Further, there is usually only one transaction log used for recovery. Concurrency control and recovery techniques that are designed for centralized database systems can be used in such systems, and are preferable to techniques described in this chapter.

Each node has its own *local* transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that node. The various trans-

Telegram: @uni_k

www.konkur.in

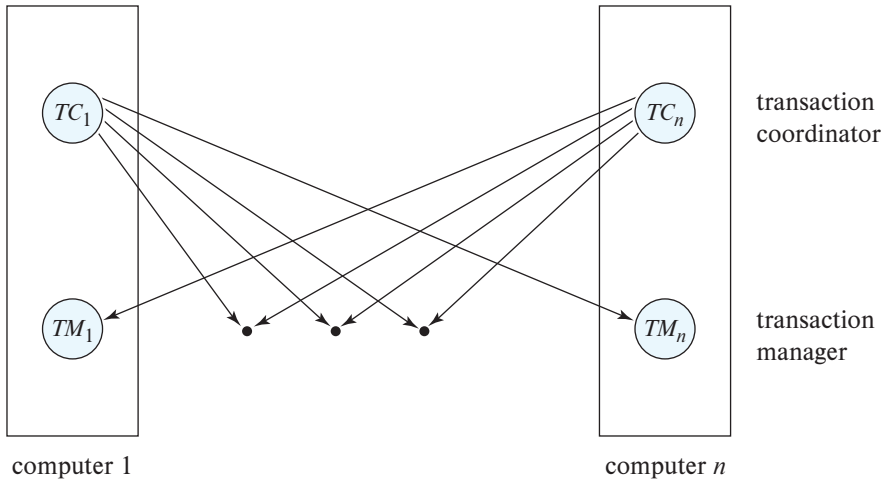


Figure 23.1 System architecture.

action managers cooperate to execute global transactions. To understand how such a manager can be implemented, consider an abstract model of a transaction system, in which each node contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in the node. Note that each such transaction may be either a local transaction (i.e., a transaction that executes at only that node) or part of a global transaction (i.e., a transaction that executes at several nodes).
- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that node.

The overall system architecture appears in Figure 23.1.

The structure of a transaction manager is similar in many respects to the structure of a centralized system. Each transaction manager is responsible for:

- Maintaining a log for recovery purposes.
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that node.

As we shall see, we need to modify both the recovery and concurrency schemes to accommodate the distributed execution of transactions.

The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single node. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that node. For each such transaction, the coordinator is responsible for:

Telegram: @uni_k

www.konkur.in

- Starting the execution of the transaction.
- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate nodes for execution.
- Coordinating the termination of the transaction, which may result in the transaction being committed at all nodes or aborted at all nodes.

23.1.2 System Failure Modes

A distributed system may suffer from the same types of failure that a centralized system does (e.g., software errors, hardware errors, or disk crashes). There are, however, additional types of failure with which we need to deal in a distributed environment. The basic failure types are:

- Failure of a node.
- Loss of messages.
- Failure of a communication link.
- Network partition.

The loss or corruption of messages is always a possibility in a distributed system. The system uses transmission-control protocols, such as TCP/IP, to handle such errors. Information about such protocols may be found in standard textbooks on networking.

However, if two nodes A and B are not directly connected, messages from one to the other must be *routed* through a sequence of communication links. If a communication link fails, messages that would have been transmitted across the link must be rerouted. In some cases, it is possible to find another route through the network, so that the messages are able to reach their destination. In other nodes, a failure may result in there being no connection between some pairs of nodes. A system is **partitioned** if it has been split into two (or more) subsystems, called **partitions**, that lack any connection between them. Note that, under this definition, a partition may consist of a single node.

23.2 Commit Protocols

If we are to ensure atomicity, all the nodes in which a transaction T executed must agree on the final outcome of the execution. T must either commit at all nodes, or it must abort at all nodes. To ensure this property, the transaction coordinator of T must execute a commit protocol.

Among the simplest and most widely used commit protocols is the **two-phase commit protocol** (**2PC**), which is described in Section 23.2.1.

Telegram: @uni_k

www.konkur.in

23.2.1 Two-Phase Commit

We first describe how the two-phase commit protocol (2PC) operates during normal operation, then describe how it handles failures and finally how it carries out recovery and concurrency control.

Consider a transaction T initiated at node N_i, where the transaction coordinator is C_i.

23.2.1.1 The Commit Protocol

When T completes its execution—that is, when all the nodes at which T has executed inform C_i that T has completed—C_i starts the 2PC protocol.

- **Phase 1**. C_i adds the record <prepare T> to the log and forces the log onto stable storage. It then sends a prepare T message to all nodes at which T executed.
  On receiving such a message, the transaction manager at that node determines whether it is willing to commit its portion of T. If the answer is no, it adds a record <no T> to the log and then responds by sending an abort T message to C_i. If the answer is yes, it adds a record <ready T> to the log and forces the log (with all the log records corresponding to T) onto stable storage. The transaction manager then replies with a ready T message to C_i.
- **Phase 2**. When C_i receives ready responses to the prepare T message from all the nodes, or when it receives an abort T message from at least one participant node, C_i can determine whether the transaction T can be committed or aborted. Transaction T can be committed if C_i received a ready T message from all the participating nodes. Otherwise, transaction T must be aborted. Depending on the verdict, either a record <commit T> or a record <abort T> is added to the log and the log is forced onto stable storage. At this point, the fate of the transaction has been sealed.

  Following this point, the coordinator sends either a commit T or an abort T message to all participating nodes. When a node receives that message, it records the result (either <commit T> or <abort T>) in its log, and correspondingly either commits or aborts the transaction.

  Since nodes may fail, the coordinator cannot wait indefinitely for responses from all the nodes. Instead, when a prespecified interval of time has elapsed since the prepare T message was sent out, if any node has not responded to the coordinator, the coordinator can decide to abort the transaction; the steps described for aborting the transaction must be followed, just as if a node had sent an abort message for the transaction.

Figure 23.2 shows an instance of successful execution of 2PC for a transaction T, with two nodes, N_1 and N_2, that are both willing to commit transaction T. If any of the

Telegram: @uni_k

www.konkur.in

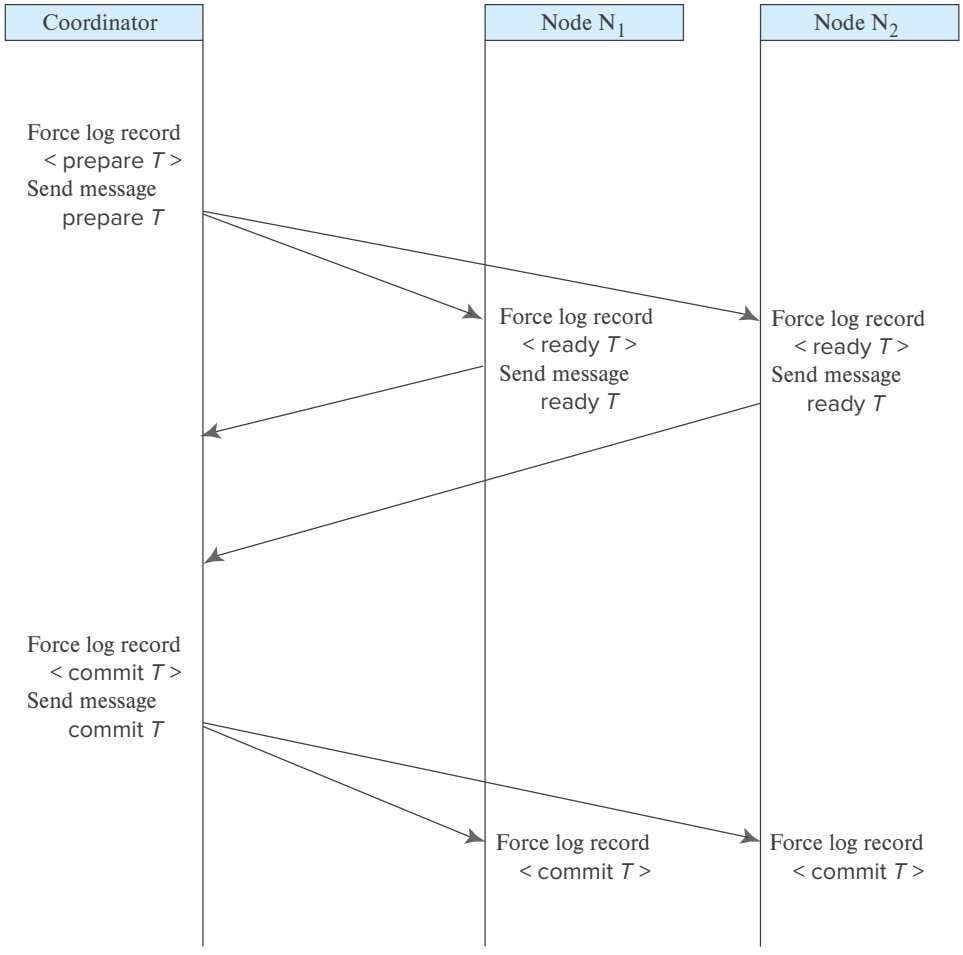


Figure 23.2 Successful execution of 2PC.

nodes sends a no T message, the coordinator will send an abort T message to all the nodes, which will then abort the transaction.

A node at which T executed can unconditionally abort T at any time before it sends the message ready T to the coordinator. Once the <ready T> log record is written, the transaction T is said to be in the **ready state** at the node. The ready T message is, in effect, a promise by a node to follow the coordinator's order to commit T or to abort T. To make such a promise, the needed information must first be stored in stable storage. Otherwise, if the node crashes after sending ready T, it may be unable to make good on its promise. Further, locks acquired by the transaction must continue to be held until the transaction completes, even if there is an intervening node failure, as we shall see in Section 23.2.1.3.

Telegram: @uni_k

www.konkur.in

Since unanimity is required to commit a transaction, the fate of T is sealed as soon as at least one node responds abort T. Since the coordinator node N_i is one of the nodes at which T executed, the coordinator can decide unilaterally to abort T. The final verdict regarding T is determined at the time that the coordinator writes that verdict (commit or abort) to the log and forces that verdict to stable storage.

In some implementations of the 2PC protocol, a node sends an acknowledge T message to the coordinator at the end of the second phase of the protocol. When the coordinator receives the acknowledge T message from all the nodes, it adds the record <complete T> to the log. Until this step, the coordinator cannot forget about the commit or abort decision on T, since a node may ask for the decision. (A node that has not received a commit or abort for transaction T, perhaps due to a network failure or temporary node failure, may send such a request to the coordinator.) After this step, the coordinator can discard information about transaction T.

23.2.1.2 Handling of Failures

The 2PC protocol responds in different ways to various types of failure:

- **Failure of a participating node**. If the coordinator C_i detects that a node has failed, it takes these actions: If the node fails before responding with a ready T message to C_i, the coordinator assumes that it responded with an abort T message. If the node fails after the coordinator has received the ready T message from the node, the coordinator executes the rest of the commit protocol in the normal fashion, ignoring the failure of the node.

  When a participating node N_k recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. Let T be one such transaction. We consider each of the possible cases:

  - The log contains a <commit T> record. In this case, the node executes redo(T).
  - The log contains an <abort T> record. In this case, the node executes undo(T).
  - The log contains a <ready T> record. In this case, the node must consult C_i to determine the fate of T. If C_i is up, it notifies N_k regarding whether T committed or aborted. In the former case, it executes redo(T); in the latter case, it executes undo(T). If C_i is down, N_k must try to find the fate of T from other nodes. It does so by sending a querystatus T message to all the nodes in the system. On receiving such a message, a node must consult its log to determine whether T has executed there, and if T has, whether T committed or aborted. It then notifies N_k about this outcome. If no node has the appropriate information (i.e., whether T committed or aborted), then N_k can neither abort nor commit T. The decision concerning T is postponed until N can obtain the needed

Telegram: @uni_k

www.konkur.in

information. Thus, N_k must periodically resend the querystatus message to the other nodes. It continues to do so until a node that contains the needed information has recovered. Note that the node at which C_i resides always has the needed information.

- The log contains no control records (abort, commit, ready) concerning T. Thus, we know that N_k failed before responding to the prepare T message from C_i. Since the failure of N_k precludes the sending of such a response, by our algorithm C_i must abort T. Hence, N_k must execute undo(T).

- **Failure of the coordinator**. If the coordinator fails in the midst of the execution of the commit protocol for transaction T, then the participating nodes must decide the fate of T. We shall see that, in certain cases, the participating nodes cannot decide whether to commit or abort T, and therefore these nodes must wait for the recovery of the failed coordinator.

  - If an active node contains a <commit T> record in its log, then T must be committed.
  - If an active node contains an <abort T> record in its log, then T must be aborted.
  - If some active node does *not* contain a <ready T> record in its log, then the failed coordinator C_i cannot have decided to commit T, because a node that does not have a <ready T> record in its log cannot have sent a ready T message to C_i. However, the coordinator may have decided to abort T, but not to commit T. Rather than wait for C_i to recover, it is preferable to abort T.
  - If none of the preceding cases holds, then all active nodes must have a <ready T> record in their logs, but no additional control records (such as <abort T> or <commit T>). Since the coordinator has failed, it is impossible to determine whether a decision has been made, and if one has, what that decision is, until the coordinator recovers. Thus, the active nodes must wait for C_i to recover.

    Since the fate of T remains in doubt, T may continue to hold system resources. For example, if locking is used, T may hold locks on data at active nodes. Such a situation is undesirable, because it may be hours or days before C_i is again active. During this time, other transactions may be forced to wait for T. As a result, data items may be unavailable not only on the failed node (C_i), but on active nodes as well. This situation is called the **blocking problem**, because T is blocked pending the recovery of node C_i.

- **Network partition**. When a network partition occurs, two possibilities exist:

  1. The coordinator and all its participants remain in one partition. In this case, the failure has no effect on the commit protocol.

Telegram: @uni_k

www.konkur.in

2. The coordinator and its participants belong to several partitions. From the viewpoint of the nodes in one of the partitions, it appears that the nodes in other partitions have failed. Nodes that are not in the partition containing the coordinator simply execute the protocol to deal with the failure of the coordinator. The coordinator and the nodes that are in the same partition as the coordinator follow the usual commit protocol, assuming that the nodes in the other partitions have failed.

Thus, the major disadvantage of the 2PC protocol is that coordinator failure may result in blocking, where a decision either to commit or to abort T may have to be postponed until C_i recovers. We discuss how to remove this limitation shortly, in Section 23.2.2.

23.2.1.3 Recovery and Concurrency Control

When a failed node restarts, we can perform recovery by using, for example, the recovery algorithm described in Section 19.4. To deal with distributed commit protocols, the recovery procedure must treat **in-doubt transactions** specially; in-doubt transactions are transactions for which a <ready T> log record is found, but neither a <commit T> log record nor an <abort T> log record is found. The recovering node must determine the commit–abort status of such transactions by contacting other nodes, as described in Section 23.2.1.2.

If recovery is done as just described, however, normal transaction processing at the node cannot begin until all in-doubt transactions have been committed or rolled back. Finding the status of in-doubt transactions can be slow, since multiple nodes may have to be contacted. Further, if the coordinator has failed, and no other node has information about the commit–abort status of an incomplete transaction, recovery potentially could become blocked if 2PC is used. As a result, the node performing restart recovery may remain unusable for a long period.

To circumvent this problem, recovery algorithms typically provide support for noting lock information in the log. (We are assuming here that locking is used for concurrency control.) Instead of writing a <ready T> log record, the algorithm writes a <ready T, L > log record, where L is a list of all write locks held by the transaction T when the log record is written. At recovery time, after performing local recovery actions, for every in-doubt transaction T, all the write locks noted in the <ready T, L > log record (read from the log) are reacquired.

After lock reacquisition is complete for all in-doubt transactions, transaction processing can start at the node, even before the commit–abort status of the in-doubt transactions is determined. The commit or rollback of in-doubt transactions proceeds concurrently with the execution of new transactions. Thus, node recovery is faster and never gets blocked. Note that new transactions that have a lock conflict with any write locks held by in-doubt transactions will be unable to make progress until the conflicting in-doubt transactions have been committed or rolled back.

Telegram: @uni_k

www.konkur.in

23.2.2 Avoiding Blocking During Commit

The blocking problem of 2PC is a serious concern for system designers, since the failure of a coordinator node could lead to blocking of a transaction that has acquired locks on a frequently used data item, which in turn prevents other transactions that need to acquire a conflicting lock from completing their execution.

By involving multiple nodes in the commit decision step of 2PC, it is possible to avoid blocking as long as a majority of the nodes involved in the commit decision are alive and can communicate with each other. This is done by using the idea of fault-tolerant distributed consensus. Details of distributed consensus are discussed in detail later, in Section 23.8, but we outline the problem and sketch a solution approach below.

The **distributed consensus problem** is as follows: A set of n nodes need to agree on a decision; in this case, whether or not to commit a particular transaction. The inputs to make the decision are provided to all the nodes, and then each node votes on the decision; in the case of 2PC, the decision is on whether or not to commit a transaction. The key goal of protocols for achieving distributed consensus is that the decision should be made in such a way that all nodes will "learn" the same value for the decision (i.e., all nodes will learn that the transaction is to be committed, or all nodes will learn that the transaction is to be aborted), even if some nodes fail during the execution of the protocol, or there are network partitions. Further, the distributed consensus protocol should not block, as long as a majority of the nodes participating remain alive and can communicate with each other.

There are several protocols for distributed consensus, two of which are widely used today (Paxos and Raft). We study distributed consensus in Section 23.8. A key idea behind these protocols is the idea of a vote, which succeeds only if a majority of the participating nodes agree on a particular decision.

Given an implementation of distributed consensus, the blocking problem due to coordinator failure can be avoided as follows: Instead of the coordinator locally deciding to commit or abort a transaction, it initiates the distributed consensus protocol, requesting that the value "committed" or "aborted" be assigned to the transaction T. The request is sent to all the nodes participating in the distributed consensus, and the consensus protocol is then executed by those nodes. Since the protocol is fault tolerant, it will succeed even if some nodes fail, as long as a majority of the nodes are up and remain connected. The transaction can be declared committed by the coordinator only after the consensus protocol completes successfully.

There are two possible failure scenarios:

- *The coordinator fails at any stage before informing all participating nodes of the commit or abort status of a transaction T.*

  In this case, a new coordinator is chosen (we will see how to do so in Section 23.7). The new coordinator checks with the nodes participating in the distributed consensus to see if a decision was made, and if so informs the 2PC participants

Telegram: @uni_k

www.konkur.in

of the decision. A majority of the nodes participating in consensus must respond, to check if a decision was made or not; the protocol will not block as long as the failed/disconnected nodes are in a minority.

If no decision was made earlier for transaction T, the new coordinator again checks with the 2PC participants to check if they are ready to commit or wish to abort the transaction, and follows the usual coordinator protocol based on their responses. As before, if no response is received from a participant, the new coordinator may choose to abort T.

- *The distributed consensus protocol fails to reach a decision.*

  Failure of the protocol can occur due to the failure of some participating nodes. It could also occur because of conflicting requests, none of which gets a majority of "votes" during the consensus protocol. For 2PC, the request normally comes from a single coordinator, so such a conflict is unlikely. However, conflicting requests can arise in rare cases if a coordinator fails after sending out a commit message, but its commit message is delivered late; meanwhile, a new 2PC coordinator makes an abort decision since it could not reach some participating nodes. Even with such a conflict, the distributed consensus protocol guarantees that only one of the commit or abort requests can succeed, even in the presence of failures. But if some nodes are down, and neither the commit nor the abort request gets a majority vote from nodes participating in the distributed consensus, it is possible for the protocol to fail to reach a decision.

  Regardless of the reason, if the distributed consensus protocol fails to reach a decision, the new coordinator just re-initiates the protocol.

Note that in the event of a network partition, a node that gets disconnected from the majority of the nodes participating in consensus may not learn about a decision, even if a decision was successfully made. Thus, transactions running at such a node may be blocked.

Failure of 2PC participants could make data unavailable, in the absence of replication. Distributed consensus can also be used to keep replicas of a data item in a consistent state, as we explain later in Section 23.8.4.

The idea of distributed consensus to make 2PC nonblocking was proposed in the 1980s; it is used, for example, in the Google Spanner distributed database system.

The **three-phase commit** (**3PC**) protocol is an extension of the two-phase commit protocol that avoids the blocking problem under certain assumptions. One variant of the protocol avoids blocking as long as network partitions do not occur, but it may lead to inconsistent decisions in the event of a network partition. Extensions of the protocol that work safely under network partitioning were developed subsequently. The idea behind these extensions is similar to the majority voting idea of distributed consensus, but the protocols are specifically tailored for the task of atomic commit.

Telegram: @uni_k

www.konkur.in

23.2.3 Alternative Models of Transaction Processing

With two-phase commit, participating nodes agree to let the coordinator decide the fate of a transaction, and are forced to wait for the decision of the coordinator, while holding locks on updated data items. While such loss of autonomy may be acceptable within an organization, no organization would be willing to force its computers to wait, potentially for a long time, while a computer at another organization makes the decision.

In this section, we describe how to use *persistent messaging* to avoid the problem of distributed commit. To understand persistent messaging, consider how one might transfer funds between two different banks, each with its own computer. One approach is to have a transaction span the two nodes and use two-phase commit to ensure atomicity. However, the transaction may have to update the total bank balance, and blocking could have a serious impact on all other transactions at each bank, since almost all transactions at the bank would update the total bank balance.

In contrast, consider how funds transfer by a bank check occurs. The bank first deducts the amount of the check from the available balance and prints out a check. The check is then physically transferred to the other bank where it is deposited. After verifying the check, the bank increases the local balance by the amount of the check. The check constitutes a message sent between the two banks. So that funds are not lost or incorrectly increased, the check must not be lost and must not be duplicated and deposited more than once. When the bank computers are connected by a network, persistent messages provide the same service as the check (but much faster).

Persistent messages are messages that are guaranteed to be delivered to the recipient exactly once (neither less nor more), regardless of failures, if the transaction sending the message commits, and are guaranteed to not be delivered if the transaction aborts. Database recovery techniques are used to implement persistent messaging on top of the normal network channels, as we shall see shortly. In contrast, regular messages may be lost or may even be delivered multiple times in some situations.

Error handling is more complicated with persistent messaging than with two-phase commit. For instance, if the account where the check is to be deposited has been closed, the check must be sent back to the originating account and credited back there. Both nodes must, therefore, be provided with error-handling code, along with code to handle the persistent messages. In contrast, with two-phase commit, the error would be detected by the transaction, which would then never deduct the amount in the first place.

The types of exception conditions that may arise depend on the application, so it is not possible for the database system to handle exceptions automatically. The application programs that send and receive persistent messages must include code to handle exception conditions and bring the system back to a consistent state. For instance, it is not acceptable to just lose the money being transferred if the receiving account has been closed; the money must be credited back to the originating account, and if that is not possible for some reason, humans must be alerted to resolve the situation manually.

Telegram: @uni_k

www.konkur.in

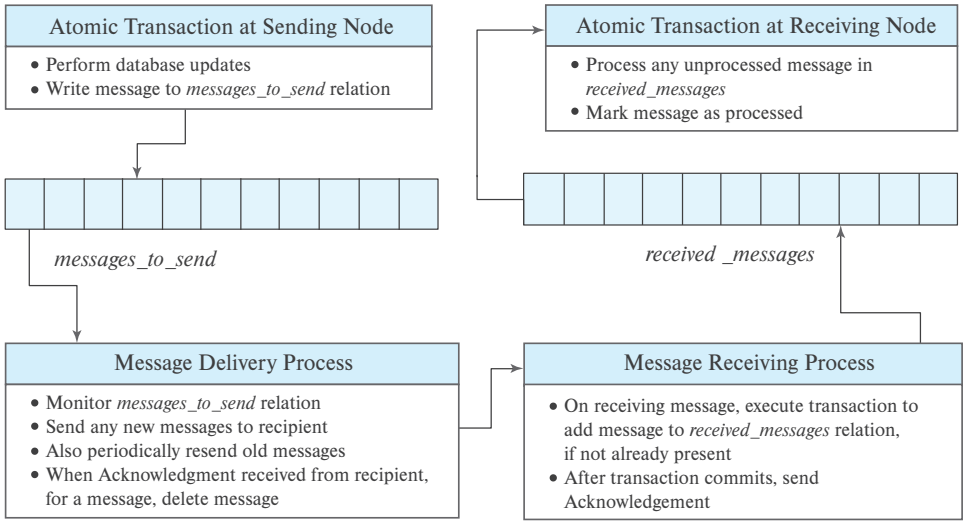


Figure 23.3 Implementation of persistent messaging.

There are many applications where the benefit of eliminating blocking is well worth the extra effort to implement systems that use persistent messages. In fact, few organizations would agree to support two-phase commit for transactions originating outside the organization, since failures could result in blocking of access to local data. Persistent messaging therefore plays an important role in carrying out transactions that cross organizational boundaries.

We now consider the **implementation** of persistent messaging. Persistent messaging can be implemented on top of an unreliable messaging infrastructure, which may lose messages or deliver them multiple times. Figure 23.3 shows a summary of the implementation, which is described in detail next.

- **Sending node protocol**. When a transaction wishes to send a persistent message, it writes a record containing the message in a special relation *messages_to_send*, instead of directly sending out the message. The message is also given a unique message identifier. Note that this relation acts as a message *outbox*.

  A *message delivery process* monitors the relation, and when a new message is found, it sends the message to its destination. The usual database concurrency-control mechanisms ensure that the system process reads the message only after the transaction that wrote the message commits; if the transaction aborts, the usual recovery mechanism would delete the message from the relation.

  The message delivery process deletes a message from the relation only after it receives an acknowledgment from the destination node. If it receives no acknowledgment from the destination node, after some time it sends the message again. It repeats this until an acknowledgment is received. In case of permanent failures,

Telegram: @uni_k

www.konkur.in

the system will decide, after some period of time, that the message is undeliverable. Exception handling code provided by the application is then invoked to deal with the failure.

Writing the message to a relation and processing it only after the transaction has committed ensures that the message will be delivered if and only if the transaction commits. Repeatedly sending it guarantees it will be delivered even if there are (temporary) system or network failures.

- **Receiving node protocol**. When a node receives a persistent message, it runs a transaction that adds the message to a special *received_messages* relation, provided it is not already present in the relation (the unique message identifier allows duplicates to be detected). The relation has an attribute to indicate if the message has been processed, which is set to false when the message is inserted in the relation. Note that this relation acts as a message *inbox*.

  After the transaction commits, or if the message was already present in the relation, the receiving node sends an acknowledgment back to the sending node.

  Note that sending the acknowledgment before the transaction commits is not safe, since a system failure may then result in loss of the message. Checking whether the message has been received earlier is essential to avoid multiple deliveries of the message.

- **Processing of message**. Received messages must be processed to carry out the actions specified in the message. A process at the receiving node monitors the *received_messages* relation to check for messages that have not been processed. When it finds such a message, the message is processed, and as part of the same transaction that processes the message, the processed flag is set to true. This ensures that a message is processed exactly once after it is received.

- **Deletion of old messages**. In many messaging systems, it is possible for messages to get delayed arbitrarily, although such delays are very unlikely. Therefore, to be safe, the message must never be deleted from the *received_messages* relation. Deleting it could result in a duplicate delivery not being detected. But as a result, the *received_messages* relation may grow indefinitely. To deal with this problem, each message is given a timestamp, and if the timestamp of a received message is older than some cutoff, the message is discarded. All messages recorded in the *received_messages* relation that are older than the cutoff can be deleted.

Workflows provide a general model of distributed transaction processing involving multiple nodes and possibly human processing of certain steps, and they are supported by application software used by enterprises. For instance, as we saw in Section 9.6.1, when a bank receives a loan application, there are many steps it must take, including contacting external credit-checking agencies, before approving or rejecting a loan application. The steps, together, form a workflow. Persistent messaging forms the underlying basis for supporting workflows in a distributed environment.

Telegram: @uni_k

www.konkur.in

23.3 Concurrency Control in Distributed Databases

We now consider how the concurrency-control schemes discussed in Chapter 18 can be modified so that they can be used in a distributed environment. We assume that each node participates in the execution of a commit protocol to ensure global transaction atomicity.

In this section, we assume that data items are not replicated, and we do not consider multiversion techniques. We discuss how to handle replicas later, in Section 23.4, and we discuss distributed multiversion concurrency control techniques in Section 23.5.

23.3.1 Locking Protocols

The various locking protocols described in Chapter 18 can be used in a distributed environment. We discuss implementation issues in this section.

23.3.1.1 Single Lock-Manager Approach

In the **single lock-manager** approach, the system maintains a *single* lock manager that resides in a *single* chosen node—say N_i. All lock and unlock requests are made at node N_i. When a transaction needs to lock a data item, it sends a lock request to N_i. The lock manager determines whether the lock can be granted immediately. If the lock can be granted, the lock manager sends a message to that effect to the node at which the lock request was initiated. Otherwise, the request is delayed until it can be granted, at which time a message is sent to the node at which the lock request was initiated. The transaction can read the data item from *any* one of the nodes at which a replica of the data item resides. In the case of a write, all the nodes where a replica of the data item resides must be involved in the writing.

The scheme has these advantages:

- **Simple implementation**. This scheme requires two messages for handling lock requests and one message for handling unlock requests.
- **Simple deadlock handling**. Since all lock and unlock requests are made at one node, the deadlock-handling algorithms discussed in Chapter 18 can be applied directly.

The disadvantages of the scheme are:

- **Bottleneck**. The node N_i becomes a bottleneck, since all requests must be processed there.
- **Vulnerability**. If the node N_i fails, the concurrency controller is lost. Either processing must stop, or a recovery scheme must be used so that a backup node can take over lock management from N_i, as described in Section 23.7.

Telegram: @uni_k

www.konkur.in

23.3.1.2 Distributed Lock Manager

A compromise between the advantages and disadvantages can be achieved through the **distributed-lock-manager** approach, in which the lock-manager function is distributed over several nodes.

Each node maintains a local lock manager whose function is to administer the lock and unlock requests for those data items that are stored in that node. When a transaction wishes to lock a data item Q that resides at node N_i, a message is sent to the lock manager at node N_i requesting a lock (in a particular lock mode). If data item Q is locked in an incompatible mode, then the request is delayed until it can be granted. Once it has determined that the lock request can be granted, the lock manager sends a message back to the initiator indicating that it has granted the lock request.

The distributed-lock-manager scheme has the advantage of simple implementation, and it reduces the degree to which the coordinator is a bottleneck. It has a reasonably low overhead, requiring two message transfers for handling lock requests, and one message transfer for handling unlock requests. However, deadlock handling is more complex, since the lock and unlock requests are no longer made at a single node: There may be internode deadlocks even when there is no deadlock within a single node. The deadlock-handling algorithms discussed in Chapter 18 must be modified, as we shall discuss in Section 23.3.2, to detect global deadlocks.

23.3.2 Deadlock Handling

The deadlock-prevention and deadlock-detection algorithms in Chapter 18 can be used in a distributed system, with some modifications.

Consider first the deadlock-prevention techniques, which we saw in Section 18.2.1.

- Techniques for deadlock prevention based on lock ordering can be used in a distributed system, with no changes at all. These techniques prevent cyclic lock waits; the fact that locks may be obtained at different nodes has no effect on prevention of cyclic lock waits.
- Techniques based on preemption and transaction rollback can also be used unchanged in a distributed system. In particular, the *wait-die* technique is used in several distributed systems. Recall that this technique allows older transactions to wait for locks held by younger transactions, but if a younger transaction needs to wait for a lock held by an older transaction, the younger transaction is rolled back. The transaction that is rolled back may subsequently be executed again; recall that it retains its original start time; if it is treated as a new transaction, it could be rolled back repeatedly, and starve, even as other transactions make progress and complete.
- Timeout-based schemes, too, work without any changes in a distributed system.

Telegram: @uni_k

www.konkur.in

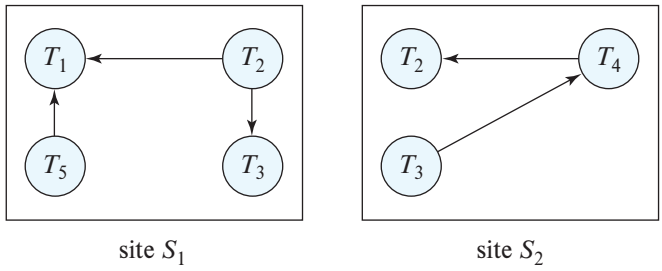


Figure 23.4 Local wait-for graphs.

Deadlock-prevention techniques may result in unnecessary waiting and rollback when used in a distributed system, just as in a centralized system,

We now consider deadlock-detection techniques that allow deadlocks to occur and detect them if they do. The main problem in a distributed system is deciding how to maintain the wait-for graph. Common techniques for dealing with this issue require that each node keep a **local wait-for graph**. The nodes of the graph correspond to all the transactions (local as well as nonlocal) that are currently either holding or requesting any of the items local to that node. For example, Figure 23.4 depicts a system consisting of two nodes, each maintaining its local wait-for graph. Note that transactions T_2 and T_3 appear in both graphs, indicating that the transactions have requested items at both nodes.

These local wait-for graphs are constructed in the usual manner for local transactions and data items. When a transaction T_i on node N_1 needs a resource in node N_2, it sends a request message to node N_2. If the resource is held by transaction T_j, the system inserts an edge $T_i \rightarrow T_j$ in the local wait-for graph of node N_2.

If any local wait-for graph has a cycle, a deadlock has occurred. On the other hand, the fact that there are no cycles in any of the local wait-for graphs does not mean that there are no deadlocks. To illustrate this problem, we consider the local wait-for graphs of Figure 23.4. Each wait-for graph is acyclic; nevertheless, a deadlock exists in the system because the *union* of the local wait-for graphs contains a cycle. This graph appears in Figure 23.5.

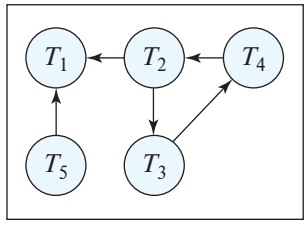


Figure 23.5 Global wait-for graph for Figure 23.4.

Telegram: @uni_k

www.konkur.in

In the **centralized deadlock detection** approach, the system constructs and maintains a **global wait-for graph** (the union of all the local graphs) in a *single* node: the deadlock-detection coordinator. Since there is communication delay in the system, we must distinguish between two types of wait-for graphs. The *real* graph describes the real but unknown state of the system at any instance in time, as would be seen by an omniscient observer. The *constructed* graph is an approximation generated by the controller during the execution of the controller's algorithm. Obviously, the controller must generate the constructed graph in such a way that, whenever the detection algorithm is invoked, the reported results are correct. *Correct* means in this case that, if a deadlock exists, it is reported promptly, and if the system reports a deadlock, it is indeed in a deadlock state.

The global wait-for graph can be reconstructed or updated under these conditions:

- Whenever a new edge is inserted in or removed from one of the local wait-for graphs.
- Periodically, when a number of changes have occurred in a local wait-for graph.
- Whenever the coordinator needs to invoke the cycle-detection algorithm.

When the coordinator invokes the deadlock-detection algorithm, it searches its global graph. If it finds a cycle, it selects a victim to be rolled back. The coordinator must notify all the nodes that a particular transaction has been selected as the victim. The nodes, in turn, roll back the victim transaction.

This scheme may produce unnecessary rollbacks if:

- **False cycles** exist in the global wait-for graph. As an illustration, consider a snapshot of the system represented by the local wait-for graphs of Figure 23.6. Suppose that T_2 releases the resource that it is holding in node N_1, resulting in the deletion of the edge $T_1 \rightarrow T_2$ in N_1. Transaction T_2 then requests a resource held by T_3 at node N_2, resulting in the addition of the edge $T_2 \rightarrow T_3$ in N_2. If the insert $T_2 \rightarrow T_3$ message from N_2 arrives before the remove $T_1 \rightarrow T_2$ message from N_1, the coordinator may discover the false cycle $T_1 \rightarrow T_2 \rightarrow T_3$ after the insert (but before the remove). Deadlock recovery may be initiated, although no deadlock has occurred.

  Note that the false-cycle situation could not occur under two-phase locking. The likelihood of false cycles is usually sufficiently low that they do not cause a serious performance problem.

- A *deadlock* has indeed occurred and a victim has been picked, while one of the transactions was aborted for reasons unrelated to the deadlock. For example, suppose that node N_1 in Figure 23.4 decides to abort T_2. At the same time, the coordinator has discovered a cycle and has picked T_3 as a victim. Both T_2 and T_3 are now rolled back, although only T_2 needed to be rolled back.

Telegram: @uni_k

www.konkur.in

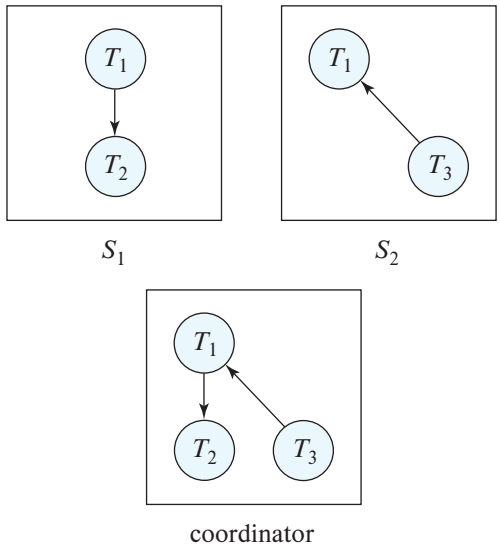


Figure 23.6 False cycles in the global wait-for graph.

Deadlock detection can be done in a distributed manner, with several nodes taking on parts of the task, instead of it being done at a single node. However, such algorithms are more complicated and more expensive. See the bibliographical notes for references to such algorithms.

23.3.3 Leases

One of the issues with using locking in a distributed system is that a node holding a lock may fail, and not release the lock. The locked data item could thus become (logically) inaccessible, until the failed node recovers and releases the lock, or the lock is released by another node on behalf of the failed node.

If an exclusive lock has been obtained on a data item, and the transaction is in the prepared state, the lock cannot be released until a commit/abort decision is made for the transaction. However, in many other cases it is acceptable for a lock that has been granted earlier to be revoked subsequently. In such cases, the concept of a lease can be very useful.

A **lease** is a lock that is granted for a specific period of time. If the process that acquires a lease needs to continue holding the lock beyond the specified period, it can **renew** the lease. A lease renewal request is sent to the lock manager, which extends the lease and responds with an acknowledgment as long as the renewal request comes in time. However, if the time expires, and the process does not renew the lease, the lease is said to **expire**, and the lock is released. Thus, any lease acquired by a node that either fails, or gets disconnected from the lock manager, is automatically released when the

Telegram: @uni_k

www.konkur.in

lease expires. The node that holds a lease regularly compares the current lease expiry time with its local clock to determine if it still has the lease or the lease has expired.

One of the uses of leases is to ensure that there is only one coordinator for a protocol in a distributed system. A node that wants to act as coordinator requests an exclusive lease on a data item associated with the protocol. If it gets the lease, it can act as coordinator until the lease expires; as long as it is active, it requests lease renewal before the lease expires, and it continues to be the coordinator as long as the lock manager permits the lease renewal.

If a node N_1 acting as coordinator dies after the expiry of the lease period, the lease automatically expires, and another node N_2 that requests the lease can acquire it and become the coordinator. In most protocols it is important that there should be only one coordinator at a given time. The lease mechanism guarantees this, as long as clocks are synchronized. However, if the coordinator's clock runs slower than the lock manager's clock, a situation can arise where the coordinator thinks it still has the lease, while the lock manager thinks the lease has expired. While clocks cannot be exactly synchronized, in practice the inaccuracy is not very high. The lock manager waits for some extra wait time after the lease expiry time to account for clock inaccuracies before it actually treats the lease as expired.

A node that checks the local clock and decides it still has a lease may then take a subsequent action as coordinator. It is possible that the lease may have expired between when the clock was checked and when the subsequent action took place, which could result in the action taking place after the node is no longer the coordinator. Further, even if the action took place while the node had a valid lease, a message sent by the node may be delivered after a delay, by which time the node may have lost its lease. While it is possible for the network to deliver a message arbitrarily late, the system can decide on a maximum message delay time, and any message that is older is ignored by the recipient; messages have timestamps set by the sender, which are used to detect if a message needs to be ignored.

The time gaps due to the above two issues can be taken into account by checking that the lease expiry is at least some time t' into the future before initiating an action, where t' is a bound on how long the action will take after the lease time check, including the maximum message delay.

We have assumed here that while coordinators may fail, the lock manager that issues leases is able to tolerate faults. We study in Section 23.8.4 how to build a fault-tolerant lock manager; we note that the techniques described in that section are general purpose and can be used to implement fault-tolerant versions of any deterministic process, modeled as a "state machine."

23.3.4 Distributed Timestamp-Based Protocols

The principal idea behind the timestamp-based concurrency control protocols in Section 18.5 is that each transaction is given a *unique* timestamp that the system uses in

Telegram: @uni_k

www.konkur.in

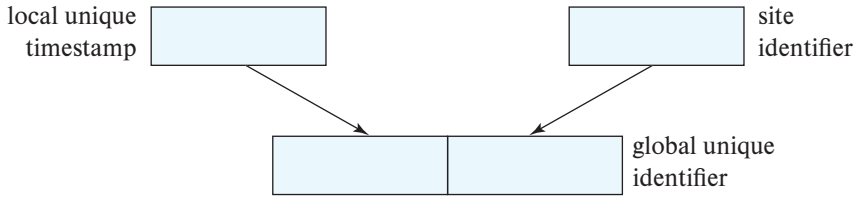


Figure 23.7 Generation of unique timestamps.

deciding the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a scheme for generating unique timestamps. We then discuss how the timestamp-based protocols can be used in a distributed setting.

23.3.5 Generation of Timestamps

There are two primary methods for generating unique timestamps, one centralized and one distributed. In the centralized scheme, a single node distributes the timestamps. The node can use a logical counter or its own local clock for this purpose. While this scheme is easy to implement, failure of the node would potentially block all transaction processing in the system.

In the distributed scheme, each node generates a unique local timestamp by using either a logical counter or the local clock. We obtain the unique global timestamp by concatenating the unique local timestamp with the node identifier, which also must be unique (Figure 23.7). If a node has multiple threads running on it (as is almost always the case today), a thread identifier is concatenated with the node identifier, to make the timestamp unique. Further, we assume that consecutive calls to get the local timestamp within a node/thread will return different timestamps; if this is not guaranteed by the local clock, the returned local timestamp value may need to be incremented, to ensure two calls do not get the same local timestamp.

The order of concatenation is important! We use the node identifier in the least significant position to ensure that the global timestamps generated in one node are not always greater than those generated in another node.

We may still have a problem if one node generates local timestamps at a rate faster than that of the other nodes. In such a case, the fast node's logical counter will be larger than that of other nodes. Therefore, all timestamps generated by the fast node will be larger than those generated by other nodes. What we need is a mechanism to ensure that local timestamps are generated fairly across the system. There are two solution approaches for this problem.

1. Keep the clocks synchronized by using a network time protocol, which is a standard feature in computers today. The protocol periodically communicates with a

Telegram: @uni_k

www.konkur.in

server to find the current time. If the local time is ahead of the time returned by the server, the local clock is slowed down, whereas if the local time is behind the time returned by the server it is speeded up, to bring it back in synchronization with the time at the server. Since all nodes are approximately synchronized with the server, they are also approximately synchronized with each other.

2. We define within each node N_i a **logical clock** (LC_i), which generates the unique local timestamp. The logical clock can be implemented as a counter that is incremented after a new local timestamp is generated. To ensure that the various logical clocks are synchronized, we require that a node N_i advance its logical clock whenever a transaction T_i with timestamp $< x, y >$ visits that node and x is greater than the current value of LC_i. In this case, node N_i advances its logical clock to the value $x + 1$. As long as messages are exchanged regularly, the logical clocks will be approximately synchronized.

23.3.6 Distributed Timestamp Ordering

The timestamp ordering protocol can be easily extended to a parallel or distributed database setting. Each transaction is assigned a globally unique timestamp at the node where it originates. Requests sent to other nodes include the transaction timestamp. Each node keeps track of the read and write timestamps of the data items at that node. Whenever an operation is received by a node, it does the timestamp checks that we saw in Section 18.5.2, locally, without any need to communicate with other nodes.

Timestamps must be reasonably synchronized across nodes; otherwise, the following problem can occur. Suppose one node has a time significantly lagging the others, and a transaction T_1 gets its timestamp at that node n_1. Suppose the transaction T_1 fails a timestamp test on a data item d_i because d_i has been updated by a transaction T_2 with a higher timestamp; T_1 would be restarted with a new timestamp, but if the time at node n_1 is not synchronized, the new timestamp may still be old enough to cause the timestamp test to fail, and T_1 would be restarted repeatedly until the time at n_1 advances ahead of the timestamp of T_2.

Note that as in the centralized case, if a transaction T_i reads an uncommitted value written by another transaction T_j, T_i cannot commit until T_j commits. This can be ensured either by making reads wait for uncommitted writes to be committed, which can be implemented using locking, or by introducing commit dependencies, as discussed in Section 18.5. The waiting time can be exacerbated by the time required to perform 2PC, if the transaction performs updates at more than one node. While a transaction T_i is in the prepared state, its writes are not committed, so any transaction with a higher timestamp that reads an item written by T_i would be forced to wait.

We also note that the *multiversion timestamp ordering protocol* can be used locally at each node, without any need to communicate with other nodes, similar to the case of the timestamp ordering protocol.

Telegram: @uni_k

www.konkur.in

23.3.7 Distributed Validation

We now consider the validation-based protocol (also called the optimistic concurrency control protocol) that we saw in Section 18.6. The protocol is based on three timestamps:

- The start timestamp StartTS(T_i).
- The validation timestamp, TS(T_i), which is used as the serialization order.
- The finish timestamp FinishTS(T_i) which identifies when the writes of a transaction have completed.

While we saw a serial version of the validation protocol in Section 18.6, where only one transaction can perform validation at a time, there are extensions to the protocol to allow validations of multiple transactions to occur concurrently, within a single system.

We now consider how to adapt the protocol to a distributed setting.

1. Validation is done locally at each node, with timestamps assigned as described below.
2. In a distributed setting, the validation timestamp TS(T_i) can be assigned at any of the nodes, but the same timestamp TS(T_i) must be used at all nodes where validation is to be performed. Transactions must be serializable based on their timestamps TS(T_i).
3. The validation test for a transaction T_i looks at all transactions T_j with TS(T_j) < TS(T_i), to check if T_j either finished before T_i started, or has no conflicts with T_i. The assumption is that once a particular transaction enters the validation phase, no transaction with a lower timestamp can enter the validation phase. The assumption can be ensured in a centralized system by assigning the timestamps in a critical section, but cannot be ensured in a distributed setting.

   A key problem in the distributed setting is that a transaction T_j may enter the validation phase after a transaction T_i, but with TS(T_j) < TS(T_i). It is too late for T_i to be validated against T_j. However, this problem can be easily fixed by rolling back any transaction if, when it starts validation at a node, a transaction with a later timestamp had already started validation at that node.
4. The start and finish timestamps are used to identify transactions T_j whose writes would definitely have been seen by a transaction T_i. These timestamps must be assigned locally at each node, and must satisfy StartTS(T_i) ≤ TS(T_i) ≤ FinishTS(T_i). Each node uses these timestamps to perform validation locally.
5. When used in conjunction with 2PC, a transaction must first be validated and then enter the prepared state. Writes cannot be committed at the database until

Telegram: @uni_k

www.konkur.in

the transaction enters the committed state in 2PC. Suppose a transaction T_j reads an item updated by a transaction T_i that is in the prepared state and is allowed to proceed using the old value of the data item (since the value generated by T_i has not yet been written to the database). Then, when transaction T_j attempts to validate, it will be serialized after T_i and will surely fail validation if T_i commits. Thus, the read by T_j may as well be held until T_i commits and finishes its writes. The above behavior is the same as what would happen with locking, with write locks acquired at the time of validation.

Although full implementations of validation-based protocols are not widely used in distributed settings, optimistic concurrency control without read validation, which we saw in Section 18.9.3, is widely used in distributed settings. Recall that the scheme depends on storing a version number with each data item, a feature that is supported by many key-value stores.[1] Version numbers are incremented each time the data item is updated.

Validation is performed at the time of writing the data item, which can be done using a test-and-set function based on version numbers, that is supported by some key-value stores. This function allows an update to a data item to be conditional on the current version of the data item being the same as a specified version number. If the current version number of the data item is more recent than the specified version number, the update is not performed. For example, a transaction that read version 7 of a data item can perform a write, conditional on the version still being at 7. If the item has been updated meanwhile, the current version would not match, and the write would fail; however, if the version number is still 7, the write would be performed successfully, and the version number incremented to 8.

The test-and-set function can thus be used by applications to implement the limited form of validation-based concurrency control, discussed in Section 18.9.3, at the level of individual data items. Thereby, a transaction could read a value from a data item, perform computation locally, and update the data item at the end, as long as the value it read has not changed subsequently. This approach does not guarantee overall serializability, but it does prevent the lost-update anomaly.

HBase supports the test-and-set operation based on comparing values (similar to the hardware test-and-set operation), which is called `checkAndPut()`. Instead of comparing to a system-generated version number, the `checkAndPut()` invocation can pass in a column and a value; the update is performed only if the row has the specified value for the specified column. The check and the update are performed atomically. A variant, `checkAndMutate()`, allows multiple modifications to a row, such as adding or updating a column, deleting a column, or incrementing a column, after checking a condition, as a single atomic operation.

[1]Note that this is not the same as multiversioning, since only one version

Telegram: @uni_k

www.konkur.in

23.4 Replication

One of the goals in using distributed databases is **high availability**; that is, the database must function almost all the time. In particular, since failures are more likely in large distributed systems, a distributed database must continue functioning even when there are various types of failures. The ability to continue functioning even during failures is referred to as **robustness**.

For a distributed system to be robust, data must be replicated, allowing the data to be accessible even if a node containing a replica of the data fails.

The database system must keep track of the locations of the replicas of each data item in the database catalog. Replication can be at the level of individual data items, in which the catalog will have one entry for each data item, recording the nodes where it is replicated. Alternatively, replication can be done at the level of partitions of a relation, with an entire partition replicated at two or more nodes. The catalog would then have one entry for each partition, resulting in considerably lower overhead than having one entry for each data item.

In this section we first discuss (in Section 23.4.1) issues with consistency of values between replicas. We then discuss (in Section 23.4.2) how to extend concurrency control techniques to deal with replicas, ignoring the issue of failures. Further extensions of the techniques to handle failures but modifying how reads and writes are executed are described in Section 23.4.3.

23.4.1 Consistency of Replicas

Given that a data item (or partition) is replicated, the system should ideally ensure that the copies have the same value. Practically, given that some nodes may be disconnected or may have failed, it is impossible to ensure that all copies have the same value. Instead, the system must ensure that even if some replicas do not have the latest value, reads of a data item get to see the latest value that was written.

More formally, the implementations of read and write operations on the replicas of a data item must follow a protocol that ensures the following property, called **linearizability**: Given a set of read and write operations on a data item,

1. there must be a linear ordering of the operations such that each read in the ordering should see the value written by the most recent write preceding the read (or the initial value if there is no such write), and
2. if an operation o_1 finishes before an operation o_2 begins (based on external time), then o_1 must precede o_2 in the linear order.

Note that linearizability only addresses what happens to a single data item, and it is orthogonal to serializability.

Telegram: @uni_k

www.konkur.in

We first consider approaches that write all copies of a data item and discuss limitations of this approach; in particular, to ensure availability during failure, failed nodes need to be removed from the set of replicas, which can be quite tricky as we will see.

It is not possible, in general, to differentiate between node failure and network partition. The system can usually detect that a failure has occurred, but it may not be able to identify the type of failure. For example, suppose that node N_1 is not able to communicate with N_2. It could be that N_2 has failed. However, another possibility is that the link between N_1 and N_2 has failed, resulting in network partition. The problem is partly addressed by using multiple links between nodes, so that even if one link fails the nodes will remain connected. However, multiple link failures can still occur, so there are situations where we cannot be sure whether a node failure or network partition has occurred.

There are protocols for data access that can continue working even if some nodes have failed, without any explicit actions to deal with the failures, as we shall see in Section 23.4.3.1. These protocols are based on ensuring a majority of nodes are written/read. With such protocols, actions to detect failed nodes and remove them from the system can be done in the background, and (re)integration of new or recovered nodes into the system can also be done without disrupting processing.

Although traditional database systems place a premium on consistency, there are many applications today that value availability more than consistency. The design of replication protocols is different for such systems and is discussed in Section 23.6.

In particular, one such alternative that is widely used for maintaining replicated data is to perform the update on a primary copy of the data item, and allow the transaction to commit without updating the other copies. However, the update is subsequently propagated to the other copies. Such propagation of updates, referred to as *asynchronous replication* or *lazy propagation of updates*, is discussed in Section 23.6.2.

One drawback of asynchronous replication is that replicas may be out of date for some time following each update. Another drawback is that if the primary copy fails after a transaction commits, but before the updates were propagated to the replicas, the updates of the committed transaction may not be visible to subsequent transactions, leading to an inconsistency.

On the other hand, a major benefit of asynchronous replication is that exclusive locks can be released as soon as the transaction commits on the primary copy. In contrast, if other replicas have to be updated before the transaction commits, there may be a significant delay in committing the transaction. In particular, if data is geographically replicated to ensure availability despite failure of an entire data center, the network round-trip time to a remote data center could range from tens of milliseconds to nearby locations, up to hundreds of milliseconds for data centers that are on the other side of the world. If a transaction were to hold a lock on a data item for this duration, the number of transactions that can update that data item would be limited to approximately 10 to 100 transactions per second. For certain applications, for example, user data in a web application, 10 to 100 transactions per second for a single data item is quite sufficient. However, for applications where some data items are updated

Telegram: @uni_k

www.konkur.in

by a large number of transactions each second, holding locks for such a long time is not acceptable. Asynchronous replication may be preferred in such cases.

23.4.2 Concurrency Control with Replicas

We discuss several alternative ways of dealing with locking in the presence of replication of data items, in Section 23.4.2.1 to Section 23.4.2.4.

In this section, we assume updates are done on all replicas of a data item. If any node containing a replica of a data item has failed, or is disconnected from the other nodes, that replica cannot be updated. We discuss how to perform reads and updates in the presence of failures later, in Section 23.4.3.

23.4.2.1 Primary Copy

When a system uses data replication, we can choose one of the replicas of a data item as the **primary copy**. For each data item Q, the primary copy of Q must reside in precisely one node, which we call the **primary node** of Q.

When a transaction needs to lock a data item Q, it requests a lock at the primary node of Q. As before, the response to the request is delayed until it can be granted. The primary copy enables concurrency control for replicated data to be handled like that for unreplicated data. This similarity allows for a simple implementation. However, if the primary node of Q fails, lock information for Q would be lost, and Q would be inaccessible, even though other nodes containing a replica may be accessible.

23.4.2.2 Majority Protocol

The **majority protocol** works this way: If data item Q is replicated in n different nodes, then a lock-request message must be sent to more than one-half of the n nodes in which Q is stored. Each lock manager determines whether the lock can be granted immediately (as far as it is concerned). As before, the response is delayed until the request can be granted. The transaction does not operate on Q until it has successfully obtained a lock on a majority of the replicas of Q.

We assume for now that writes are performed on all replicas, requiring all nodes containing replicas to be available. However, the major benefit of the majority protocol is that it can be extended to deal with node failures, as we shall see in Section 23.4.3.1. The protocol also deals with replicated data in a decentralized manner, thus avoiding the drawbacks of central control. However, it suffers from these disadvantages:

- **Implementation**. The majority protocol is more complicated to implement than are the previous schemes. It requires at least $2(n/2 + 1)$ messages for handling lock requests and at least $(n/2 + 1)$ messages for handling unlock requests.
- **Deadlock handling**. In addition to the problem of global deadlocks due to the use of a distributed-lock-manager approach, it is possible for a deadlock to occur even if only one data item is being locked. As an illustration, consider a system with four nodes and full replication. Suppose

Telegram: @uni_k

www.konkur.in

data item Q in exclusive mode. Transaction T_1 may succeed in locking Q at nodes N_1 and N_3, while transaction T_2 may succeed in locking Q at nodes N_2 and N_4. Each then must wait to acquire the third lock; hence, a deadlock has occurred. Luckily, we can avoid such deadlocks with relative ease by requiring all nodes to request locks on the replicas of a data item in the same predetermined order.

23.4.2.3 Biased Protocol

The **biased protocol** is another approach to handling replication. The difference from the majority protocol is that requests for shared locks are given more favorable treatment than requests for exclusive locks.

- **Shared locks**. When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one node that contains a replica of Q.
- **Exclusive locks**. When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all nodes that contain a replica of Q.

As before, the response to the request is delayed until it can be granted.

The biased scheme has the advantage of imposing less overhead on `read` operations than does the majority protocol. This savings is especially significant in common cases in which the frequency of `read` is much greater than the frequency of `write`. However, the additional overhead on writes is a disadvantage. Furthermore, the biased protocol shares the majority protocol's disadvantage of complexity in handling deadlock.

23.4.2.4 Quorum Consensus Protocol

The **quorum consensus** protocol is a generalization of the majority protocol. The quorum consensus protocol assigns each node a nonnegative weight. It assigns read and write operations on an item x two integers, called **read quorum** Q_r and **write quorum** Q_w, that must satisfy the following condition, where S is the total weight of all nodes at which x resides:

$$Q_r + Q_w > S \textbf{ and } 2 * Q_w > S$$

To execute a read operation, enough replicas must be locked that their total weight is at least Q_r. To execute a write operation, enough replicas must be locked so that their total weight is at least Q_w.

A benefit of the quorum consensus approach is that it can permit the cost of either read or write locking to be selectively reduced by appropriately defining the read and write quorums. For instance, with a small read quorum, reads need to obtain fewer locks, but the write quorum will be higher, hence writes need to obtain more locks. Also, if higher weights are given to some nodes (e.g., those less likely to fail), fewer

Telegram: @uni_k

www.konkur.in

nodes need to be accessed for acquiring locks. In fact, by setting weights and quorums appropriately, the quorum consensus protocol can simulate the majority protocol and the biased protocols.

Like the majority protocol, quorum consensus can be extended to work even in the presence of node failures, as we shall see in Section 23.4.3.1.

23.4.3 Dealing with Failures

Consider the following protocol to deal with replicated data. Writes must be successfully performed at all replicas of a data item. Reads may read from any replica. When coupled with two-phase locking, such a protocol will ensure that reads will see the value written by the most recent write to the same data item. This protocol is also called the **read one, write all copies** protocol since all replicas must be written, and any replica can be read.

The problem with this protocol lies in what to do if some node is unavailable. To allow work to proceed in the event of failures, it may appear that we can use a "read one, write all available" protocol. In this approach, a read operation proceeds as in the **read one, write all** scheme; any available replica can be read, and a read lock is obtained at that replica. A write operation is shipped to all replicas, and write locks are acquired on all the replicas. If a node is down, the transaction manager proceeds without waiting for the node to recover. While this approach appears very attractive, it does not guarantee consistency of writes and reads. For example, a temporary communication failure may cause a node to appear to be unavailable, resulting in a write not being performed, but when the link is restored, the node is not aware that it has to perform some reintegration actions to catch up on writes it has lost. Further, if the network partitions, each partition may proceed to update the same data item, believing that nodes in the other partitions are all dead.

23.4.3.1 Robustness Using the Majority-Based Protocol

The majority-based approach to distributed concurrency control in Section 23.4.2.2 can be modified to work in spite of failures. In this approach, each data object stores with it a version number to detect when it was last written. Whenever a transaction writes an object it also updates the version number in this way:

- If data object a is replicated in n different nodes, then a lock-request message must be sent to more than one-half of the n nodes at which a is stored. The transaction does not operate on a until it has successfully obtained a lock on a majority of the replicas of a.

  Updates to the replicas can be committed atomically using 2PC. (We assume for now that all replicas that were accessible stay accessible until commit, but we

Telegram: @uni_k

www.konkur.in

relax this requirement later in this section, where we also discuss alternatives to 2PC.)

- Read operations look at all replicas on which a lock has been obtained and read the value from the replica that has the highest version number. (Optionally, they may also write this value back to replicas with lower version numbers.) Writes read all the replicas just like reads to find the highest version number (this step would normally have been performed earlier in the transaction by a read, and the result can be reused). The new version number is one more than the highest version number. The write operation writes all the replicas on which it has obtained locks and sets the version number at all the replicas to the new version number.

Failures (whether network partitions or node failures) can be tolerated as long as (1) the nodes available at commit contain a majority of replicas of all the objects written to and (2) during reads, a majority of replicas are read to find the version numbers. If these requirements are violated, the transaction must be aborted. As long as the requirements are satisfied, the two-phase commit protocol can be used, as usual, on the nodes that are available.

In this scheme, reintegration is trivial; nothing needs to be done. This is because writes would have updated a majority of the replicas, while reads will read a majority of the replicas and find at least one replica that has the latest version.

However, the majority protocol using version numbers has some limitations, which can be avoided by using extensions or by using alternative protocols.

1. The first problem is how to deal with the failure of participants during an execution of the two-phase commit protocol.

   This problem can be dealt with by an extension of the two-phase commit protocol that allows commit to happen even if some replicas are unavailable, as long as a majority of replicas of a partition confirm that they are in prepared state. When participants recover or get reconnected, or otherwise discover that they do not have the latest updates, they need to query other nodes to catch up on missing updates. References that provide details of such solutions may be found in the bibliographic notes for this chapter, available online.

2. The second problem is how to deal with the failure of the coordinator during an execution of two-phase commit protocol, which could lead to the blocking problem. Consensus protocols, which we study in Section 23.8, provide a robust way of implementing two-phase commit without the risk of blocking even if the coordinator fails, as long as a majority of the nodes are up and connected, as we will see in Section 23.8.5.

3. The third problem is that reads pay a higher price, having to contact a majority of the copies. We study approaches to reducing the read overhead in Section 23.4.3.2.

Telegram: @uni_k

www.konkur.in

23.4.3.2 Reducing Read Cost

One approach to dealing with this problem is to use the idea of read and write quorums from the quorum consensus protocol; reads can read from a smaller read quorum, while writes have to successfully write to a larger write quorum. There is no change to the version numbering technique described earlier. The drawback of this approach is that a higher write quorum increases the chance of blocking of update transactions, due to failure or disconnection of nodes. As a special case of quorum consensus, we give unit weights to all nodes, set the read quorum to 1, and set the write quorum to n (all nodes). This corresponds to the read-any-write-all protocol we saw earlier. There is no need to use version numbers with this protocol. However, if even a single node containing a data item fails, no write to the item can proceed, since the write quorum will not be available.

A second approach is to use the primary copy technique for concurrency control and force all updates to go through the primary copy. Reads can be satisfied by accessing only one node, in contrast to the majority or quorum protocols. However, an issue with this approach is how to handle failures. If the primary copy node fails, and another node is assigned to act as the primary copy, it must ensure that it has the latest version of all data items. Subsequently, reads can be done at the primary copy, without having to read data from other copies.

This approach requires that there be at most one node that can act as primary copy at a time, even in the event of network partitions. This can be ensured using leases as we saw earlier in Section 23.7. Furthermore, this approach requires an efficient way for the new coordinator to ensure that it has the latest version of all data items. This can be done by having a log at each node and ensuring the logs are consistent with each other. This problem is by itself a nontrivial process, but it can be solved using distributed consensus protocols which we study in Section 23.8. Distributed consensus internally uses a majority scheme to ensure consistency of the logs. But it turns out that if distributed consensus is used to keep logs synchronized, there is no need for version numbering.

In fact, consensus protocols provide a way of implementing fault-tolerant replication of data, as we see later in Section 23.8.4. Many fault-tolerant storage system implementations today are built using fault-tolerant replication of data based on consensus protocols.

There is a variant of the primary copy scheme, called the **chain replication** protocol, where the replicas are ordered. Each update is sent to the first replica, which records it locally and forwards it to the next replica, and so on. The update is completed when the last (tail) replica receives the update. Reads must be executed at the tail replica, to ensure that only updates that have been fully replicated are read. If a node in a replica chain fails, reconfiguration is required to update the chain; further, the system must ensure that any incomplete updates are completed before processing further updates. Optimized versions of the chain replication scheme are used in several storage systems.

Telegram: @uni_k

www.konkur.in

References providing more details of the chain replication protocol may be found in the Further Reading section at the end of the chapter.

23.4.4 Reconfiguration and Reintegration

While nodes do fail, in most cases nodes recover soon, and the protocols described earlier can ensure that they will catch up with any updates that they missed.

However, in some cases a node may fail permanently. The system must then be **reconfigured** to remove failed nodes, and to allow other nodes to take over the tasks assigned to the failed node. Further, the database catalog must be updated to remove the failed node from the list of replicas of all data items (or relation partitions) that were replicated at that node.

As discussed earlier, a network failure may result in a node appearing to have failed, even if it has not actually failed. It is safe to remove such a node from the list of replicas; reads will no longer be routed to the node even though it may be accessible, but that will not cause any consistency problems.

If a failed node that was removed from the system eventually recovers, it must be **reintegrated** into the system. When a failed node recovers, if it had replicas of any partition or data item, it must obtain the current values of these data items it stores. The database recovery log at a live site can be used to find and perform all updates that happened when the node was down,

Reintegration of a node is more complicated than it may seem to be at first glance, since there may be updates to the data items processed during the time that the node is recovering. The database recovery log at a live site is used for catching up with the latest values for all data items at the node. Once it has caught up with the current value of all data items, the node should be added back into the list of replicas for the relevant partitions/data items, so it will receive all future updates. Locks are obtained on the partitions/data items, updates up to that point are applied from the log, and the node is added to the list of replicas for the partitions or data items, before releasing the locks. Subsequent updates will be applied directly to the node, since it will be in the list of replicas.

Reintegration is much easier with the majority-based protocols in Section 23.4.3.1, since the protocol can tolerate nodes with out-of-date data. In this case, a node can be reintegrated even before catching up on updates, and the node can catch up with missed updates subsequently.

Reconfiguration depends on nodes having an up-to-date version of the catalog that records what table partitions (or data items) are replicated at what nodes; thus information must be consistent across all nodes in a system. The replication information in the catalog could be stored centrally, and consulted on each access, but such a design would not be scalable since the central node would be consulted very frequently and would get overloaded. To avoid such a bottleneck, the catalog itself needs to be partitioned, and it may be replicated, for example, using the majority protocol.

Telegram: @uni_k

www.konkur.in

23.5 Extended Concurrency Control Protocols

In this section, we describe further extensions to distributed concurrency control protocols. We first consider multiversion 2PL and how it can be extended to get globally consistent timestamps, in Section 23.5.1. Extensions of snapshot isolation to distributed settings are described in Section 23.5.2. Issues in concurrency control in heterogeneous distributed databases, where each node may have its own concurrency control technique, are described in Section 23.5.3.

23.5.1 Multiversion 2PL and Globally Consistent Timestamps

The multiversion two-phase locking (MV2PL) protocol, described in Section 18.7.2, combines the benefits of lock-free read-only transactions with the serializability guarantees of two-phase locking. Read-only transactions see a snapshot at a point in time, while update transactions use two-phase locking but create new versions of each data item that they update. Recall that with this protocol, each transaction T_i gets a unique timestamp CommitTS(T_i) (which could be a counter, instead of actual time) at the time of commit. The transaction sets the timestamp of all items that it updates to CommitTS(T_i). Only one transaction performs commit at a point in time; this guarantees that once T_i commits, a read-only transaction T_j whose StartTS(T_j) is set to CommitTS(T_i) will see committed values of all versions with timestamp $\leq$ CommitTS(T_i).

MV2PL can be extended to work in a distributed setting by having a central coordinator, that assigns start and commit timestamps and ensures that only one transaction can perform commit at a point in time. However, the use of a central coordinator limits scalability in a massively parallel data storage system.

The Google Spanner data storage system pioneered a version of the MV2PL system that is scalable and uses timestamps based on real clock time. We study the Spanner MV2PL implementation in the rest of this section.

Suppose every node has a perfectly accurate clock, and that commit processing can happen instantly with no delay between initiation of commit and its completion. Then, when a transaction wants to commit, it gets a commit timestamp by just reading the clock at any one node at any time after getting all locks, but before releasing any lock. All data item versions created by the transaction use this commit timestamp. Transactions can be serialized by this commit timestamp. Read-only transactions simply read the clock when they start and use it to get a snapshot of the database as of their start time.

If the clocks are perfectly accurate, and commit processing is instantaneous, this protocol can be used to implement MV2PL without any central coordination, making it very scalable.

Unfortunately, in the real world, the above assumptions do not hold, which can lead to the following problems:

Telegram: @uni_k

www.konkur.in

1. Clocks are never perfectly accurate, and the clock at each node may be a little fast or a little slow compared to other clocks.

   Thus, it is possible to have the following situation. Two update transactions T_1 and T_2, both write a data item x, with T_1 writing it first, followed by T_2 but T_2 may end up with a lower commit timestamp because it got the timestamp at a different node than T_1. This situation is not consistent with the serialization ordering of T_1 and T_2, and it cannot happen with MV2PL in a centralized setting.

2. Commit processing takes time, which can cause read-only transactions to miss updates if the protocol is not carefully designed. Consider the following situation. A read-only transaction T_1 with start timestamp t_1 reads data item x at node N_1, it is possible that soon after the read, another transaction T_2 with CommitTS(T_2) $\leq$ t_1 (which got the commit timestamp at a different node N_2) may still perform a write on x. Then, T_1 should have read the value written by T_2, but did not see it.

To deal with the first problem, namely, the lack of clock synchronization, Spanner uses the following techniques.

- Spanner has a few atomic clocks that are very accurate at each data center and uses the time they provide, along with time information from the Global Positioning system (GPS) satellites, which provides very accurate time information, to get a very good estimate of time at each node. We use the term *true time* to refer to the time that would have been given by an absolutely accurate clock.

  Each node periodically communicates with time servers to synchronize its clock; if the clock has gone faster it is (logically) slowed down, and if it is slower, it is moved forward to the time from the server. In between synchronizations the local clock continues to tick, advancing the local time. A clock that ticks slower or faster than the correct rate results in local time at the node that is progressively behind or ahead of the true time.

- The second key technique is to measure local clock drift each time the node synchronizes with a time server and to use it to estimate the rate at which the local clock loses or gains time. Using this information, the Spanner system maintains a value ϵ such that if the local clock time is t', the true time t is bounded by $t' - \epsilon \leq t \leq t' + \epsilon$. The Spanner system is able to keep the uncertainty value ϵ to less than 10 msec typically. The TrueTime API used by Spanner allows the system to get the current time value, along with an upper bound on the uncertainty in the time value.

- The next piece of the solution is an idea called **commit wait**. The idea is as follows: After all locks have been acquired at all nodes, the local time t' is read at a coordinator node. We would like to use the true time as timestamp, but we don't have the exact value. Instead, the highest possible value of true time, namely, $t' + \epsilon$, is used as a commit timestamp t_c. The transaction then waits, *while holding locks*, until it

Telegram: @uni_k

www.konkur.in

is sure that the true time t is $\geq t_c$; this just requires waiting for a time interval 2ϵ, calculated as described earlier.

What the commit wait guarantees is that if a transaction T_1 has a commit timestamp t_c, at the true time t_c all locks were held by T_1.

- Given the above, if a version x_t of a data item x has a timestamp t, we can say that that was indeed the value of x at true time t. This allows us to define a snapshot of the database at a time t, containing the latest versions of all data items as of time t. A database system is said to provide **external consistency** if the serialization order is consistent with the real-world time ordering in which the transactions commit. Spanner guarantees external consistency by ensuring that the timestamps used to define the transaction serialization order correspond to the true time when the transactions commit.

- One remaining issue is that transaction commit processing takes time (particularly so when 2PC is used). While a transaction with commit timestamp t is committing, a read of x by a read-only transaction with timestamp $t_1 \geq t$ may not see the version x_t, either because the timestamp has not yet been propagated to the node with the data item x, or the transaction is in prepared state.

  To deal with this problem, reads that ask for a snapshot as of time t_1 are made to wait until the system is sure that no transactions with timestamp $\leq t_1$ are still in the process of committing. If a transaction with timestamp $t \leq t_1$ is currently in the prepared phase of 2PC, and we are not sure whether it will commit or abort, a read with timestamp t_1 would have to wait until we know the final commit status of the transaction.

  Read-only transactions can be given a somewhat earlier timestamp, to guarantee that they will not have to wait; the trade-off here is that to avoid waiting, the transaction may not see the latest version of some data items.

23.5.2 Distributed Snapshot Isolation

Since snapshot isolation is widely used, extending it to work in a distributed setting is of significant practical importance. Recall from Section 18.8 that while snapshot isolation does not guarantee serializability, it avoids a number of concurrency anomalies.

If each node implements snapshot isolation independently, the resultant schedules can have anomalies that cannot occur in a centralized system. For example, suppose two transactions, T_1 and T_2 run concurrently on node N_1, where T_1 writes x and T_2 reads x; thus T_2 would not see updates made by T_1 to x. Suppose also that T_1 updates a data item y at node N_2, and commits, and subsequently T_2 reads y at node N_2. Then T_2 would see the value of y updated by T_1 at N_2, but not see T_1's update to x at N_1. Such a situation could never occur when using snapshot isolation at a single node. Thus, just depending on local enforcement of snapshot isolation at each node is not sufficient to enforce snapshot isolation across nodes.

Telegram: @uni_k

www.konkur.in

Several alternative distributed snapshot isolation protocols have been proposed in the literature. Since the protocols are somewhat complicated, we omit details, but references with more details may be found in the bibliographic notes for this chapter, available online. Some of these protocols allow local transactions at each node to execute without any global coordination step; an extra cost is paid only by global transactions, that is, transactions that execute at more than one node. These protocols have been prototyped on several databases/data storage systems, such as SAP HANA and HBase.

There has been some work on extending distributed snapshot isolation protocols to make them serializable. Approaches explored include adding timestamp checks similar to timestamp ordering, creating a transaction dependency graph at a central server, and checking for cycles in the graph, among other approaches.

23.5.3 Concurrency Control in Federated Database Systems

Recall from Section 20.5 that in many cases a distributed database has to be constructed by linking together multiple already-existing database systems, each with its own schema and possibly running different database-management software. Recall that such systems are called *federated database systems* or *heterogeneous distributed database systems*, and they consist of a layer of software on top of the existing database systems.

Transactions in a federated database may be classified as follows:

1. **Local transactions**. These transactions are executed by each local database system outside of the federated database system's control.
2. **Global transactions**. These transactions are executed under the control of the federated database system.

The federated database system is aware of the fact that local transactions may run at the local nodes, but it is not aware of what specific transactions are being executed, or of what data they may access.

Ensuring the local autonomy of each database system requires that no changes be made to its software. A database system at one node thus is not able to communicate directly with one at any other node to synchronize the execution of a global transaction active at several nodes.

Since the federated database system has no control over the execution of local transactions, each local system must use a concurrency-control scheme (e.g., two-phase locking or timestamping) to ensure that its schedule is serializable. In addition, in the case of locking, the local system must be able to guard against the possibility of local deadlocks.

The guarantee of local serializability is not sufficient to ensure global serializability. As an illustration, consider two global transactions T_1 and T_2, each of which accesses and updates two data items, A and B, located at nodes N_1 and N_2, respectively. Suppose that the local schedules are serializable. It is still possible to have a situation where, at

Telegram: @uni_k

www.konkur.in

node N_1, T_2 follows T_1, whereas, at N_2, T_1 follows T_2, resulting in a nonserializable global schedule. Indeed, even if there is no concurrency among global transactions (i.e., a global transaction is submitted only after the previous one commits or aborts), local serializability is not sufficient to ensure global serializability (see Practice Exercise 23.11).

Depending on the implementation of the local database systems, a global transaction may not be able to control the precise locking behavior of its local subtransactions. Thus, even if all local database systems follow two-phase locking, it may be possible only to ensure that each local transaction follows the rules of the protocol. For example, one local database system may commit its subtransaction and release locks, while the subtransaction at another local system is still executing. If the local systems permit control of locking behavior and all systems follow two-phase locking, then the federated database system can ensure that global transactions lock in a two-phase manner and the lock points of conflicting transactions would then define their global serialization order. If different local systems follow different concurrency-control mechanisms, however, this straightforward sort of global control does not work.

There are many protocols for ensuring consistency despite the concurrent execution of global and local transactions in federated database systems. Some are based on imposing sufficient conditions to ensure global serializability. Others ensure only a form of consistency weaker than serializability but achieve this consistency by less restrictive means.

There are several schemes to ensure global serializability in an environment where update transactions as well as read-only transactions can execute. Several of these schemes are based on the idea of a **ticket**. A special data item called a ticket is created in each local database system. Every global transaction that accesses data at a node must write the ticket at that node. This requirement ensures that global transactions conflict directly at every node they visit. Furthermore, the global transaction manager can control the order in which global transactions are serialized, by controlling the order in which the tickets are accessed. References to such schemes appear in the bibliographic notes for this chapter, available online.

23.6 Replication with Weak Degrees of Consistency

The replication protocols we have seen so far guarantee consistency, even if there are node and network failures. However, these protocols have a nontrivial cost, and further they may block if a significant number of nodes fail or get disconnected due to a network partition. Further, in the case of a network partition, a node that is not in the majority partition would not only be unable to perform writes, but it would also be unable to perform even reads.

Telegram: @uni_k

www.konkur.in

Many applications wish to have higher availability, even at the cost of consistency. We study the trade-offs between consistency and availability in this section.

23.6.1 Trading Off Consistency for Availability

The protocols we have seen so far require a (weighted) majority of nodes be in a partition for updates to proceed. Nodes that are in a minority partition cannot process updates; if a network failure results in more than two partitions, no partition may have a majority of nodes. Under such a situation, the system would be completely unavailable for updates, and depending on the read-quorum, may even become unavailable for reads. The write-all-available protocol which we saw earlier provides availability but not consistency.

Ideally, we would like to have consistency and availability, even in the face of partitions. Unfortunately, this is not possible, a fact that is crystallized in the so-called **CAP theorem**, which states that any distributed database can have at most two of the following three properties:

- Consistency.
- Availability.
- Partition-tolerance.

The proof of the CAP theorem uses the following definition of consistency, with replicated data: an execution of a set of operations (reads and writes) on replicated data is said to be **consistent** if its result is the same as if the operations were executed on a single node, in a sequential order that is consistent with the ordering of operations issued by each process (transaction). The notion of consistency is similar to atomicity of transactions, but with each operation treated as a transaction, and is weaker than the atomicity property of transactions.

In any large-scale distributed system, partitions cannot be prevented, and as a result, either availability or consistency has to be sacrificed. The schemes we have seen earlier sacrifice availability for consistency in the face of partitions.

Consider a web-based social-networking system that replicates its data on three servers, and a network partition occurs that prevents the servers from communicating with each other. Since none of the partitions has a majority, it would not be possible to execute updates on any of the partitions. If one of these servers is in the same partition as a user, the user actually has access to data, but would be unable to update the data, since another user may be concurrently updating the same object in another partition, which could potentially lead to inconsistency. Inconsistency is not as great a risk in a social-networking system as in a banking database. A designer of such a system may decide that a user who can access the system should be allowed to perform updates on whatever replicas are accessible, even at the risk of inconsistency.

Telegram: @uni_k

www.konkur.in

In contrast to systems such as banking databases that require the ACID properties, systems such as the social-networking system mentioned above are said to require the **BASE** properties:

- Basically available.
- Soft state.
- Eventually consistent.

The primary requirement is availability, even at the cost of consistency. Updates should be allowed, even in the event of partitioning, following, for example, the write-all-available protocol (which is similar to multimaster replication described in Section 23.6). Soft state refers to the property that the state of the database may not be precisely defined, with each replica possibly having a somewhat different state due to partitioning of the network. Eventually consistent is the requirement that once a partitioning is resolved, eventually all replicas will become consistent with each other.

This last step requires that inconsistent copies of data items be identified; if one is an earlier version of the other, the earlier version can be replaced by the later version. It is possible, however, that the two copies were the result of independent updates to a common base copy. A scheme for detecting such inconsistent updates, called the version-vector scheme, is described in Section 23.6.4.

Restoring consistency in the face of inconsistent updates requires that the updates be merged in some way that is meaningful to the application. We discuss possible solutions for resolution of conflicting updates, in Section 23.6.5.

In general, no system designer wants to deal with the possibility of inconsistent updates and the resultant problems of detection and resolution. Where possible, the system should be kept consistent. Inconsistent updates are allowed only when a node is disconnected from the network, in applications that can tolerate inconsistency.

Some key-value stores such as Apache Cassandra and MongoDB allow an application to specify how many replicas need to be accessible to carry out a write operation or a read operation. As long as a majority of replicas are accessible, there is no problem with consistency for writes. However, if the application sets the required number at less than a majority, and many replicas are inaccessible, updates are allowed to go ahead; there is, however, a risk of inconsistent updates, which must be resolved later.

For applications where inconsistency can cause significant problems, or is harder to resolve, system designers prefer to build fault-tolerant systems using replication and distributed consensus that avoid inconsistencies, even at the cost of potential non-availability.

23.6.2 Asynchronous Replication

Many relational database systems support replication with weak consistency, which can take one of several forms.

Telegram: @uni_k

www.konkur.in

With **asynchronous replication** the database allows updates at a *primary* node (also referred to as the **master** node) and propagates updates to replicas at other nodes subsequently; the transaction that performs the update can commit once the update is performed at the primary, even before replicas are updated. Propagation of updates after commit is also referred to as **lazy propagation**. In contrast, the term **synchronous replication** refers to the case where updates are propagated to other replicas as part of a single transaction.

With asynchronous replication, the system must guarantee that once the transaction commits at the primary, the updates are eventually propagated to all replicas, even if there are system failures in between. Later in this section, we shall see how this property is guaranteed using persistent messaging.

Since propagation of updates is done asynchronously, a read at a replica may not get the latest version of a data item. Asynchronous propagation of updates is commonly used to allow update transactions to commit quickly, even at the cost of consistency. A system designer may choose to use replicas only for fault tolerance. However, if the replica is available on a local machine, or another machine that can be accessed with low latency, it may be much cheaper to read the data item at the replica instead of reading it from the primary, as long as the application is willing to accept potentially stale data values.

Data storage systems based on asynchronous replication may allow data items to have versions, with associated timestamps. A transaction may then request a version with required freshness properties, for example not more than 10 minutes old. If a local replica has a version of the data item satisfying the freshness criterion, it can be used; otherwise, the read may have to be sent to the primary node.

Consider, for example, an airline reservation site that shows the prices of multiple flight options. Prices may vary frequently, and the system does not guarantee that a user will actually be able to book a ticket at the price shown initially. Thus, it is quite acceptable to show a price that is a few minutes old. Asynchronous replication is a good solution for this application: price data can be replicated to a large number of servers, which share the load of user queries; and price data are updated at a primary node and replicated asynchronously to all other replicas.

Multiversion concurrency control schemes can be used to give a *transaction-consistent snapshot* of the database to read-only transactions that execute at a replica; that is, the transaction should see all updates of all transactions up to some transaction in the serialization order and should not see any updates of transactions later in the serialization order. The multiversion 2PL scheme, described in Section 23.5.1, can be extended to allow a read-only transaction to access a replica that may not have up-to-date versions of some data items, but still get a transaction-consistent snapshot view of the database. To do so, replicas must be aware of what is the latest timestamp t_{safe} such that they have received all updates with commit timestamp before t. Any read of a snapshot with timestamp $t < t_{safe}$ can be processed by that replica. Such a scheme is used in the Google Spanner database,

Telegram: @uni_k

www.konkur.in

Asynchronous replication is used in traditional (centralized) databases to create one or more replicas of the database, on which large queries can be executed, without interfering with transactions running on a primary node. Such replication is referred to **master-slave replication**, since the replicas cannot perform any updates on their own but must only perform updates that the master node asks them to perform.

In such systems, asynchronous propagation of updates is typically done in a continuous fashion to minimize delays until an update is seen at a replica. However, in data warehouses, updates may be propagated periodically—every night, for example—so that update propagation does not interfere with query processing.

Some database systems support **multimaster replication** (also called **update-anywhere replication**); updates are permitted at any replica of a data item and are propagated to all replicas either synchronously, using two-phase commit, or asynchronously.

Asynchronous replication is also used in some distributed storage systems. Such systems partition data, as we have seen earlier, but replicate each partition. There is a primary node for each partition, and updates are typically sent to the primary node, which commits the updates locally, and propagates them asynchronously to the other replicas of the partition. Some systems such as PNUTS even allow each data item in a partition to specify which node should act as the primary node for that data item; that node is responsible for committing updates to the data item, and propagating the update to the other replicas. The motivation is to allow a node that is geographically close to a user to act as the primary node for data items corresponding to that user.

In any system supporting asynchronous propagation of updates, it is important that once an update is committed at the primary, it must definitely be delivered to the other replicas. If there are multiple updates at a primary node, they must be delivered in the same order to the replicas; out-of-order delivery can cause an earlier update to arrive late and overwrite a later update.

Persistent messaging, which we saw in Section 23.2.3, provides guaranteed delivery of messages and is widely used for asynchronous replication. The implementation techniques for persistent messages described in Section 23.2.3 can be easily modified to ensure that messages are delivered in the order in which they were sent. With persistent messaging, each primary node needs to be aware of the location of all the replicas.

Publish-subscribe systems, which we saw in Section 22.8.1, offer a more flexible way of ensuring reliable message delivery. Recall that publish-subscribe systems allow messages to be published with an associated topic, and subscribers can subscribe to any desired topic. To implement asynchronous replication, a topic is created corresponding to each partition. All replicas of a partition subscribe to the topic corresponding to the partition. Any update (including inserts, deletes, and data item updates) to a partition is published as a message with the topic corresponding to the partition. The publish-subscribe system ensures that once such a message is published, it will be delivered to all subscribers in the order in which it was published.

Publish-subscribe systems designed for parallel systems, such as the Apache Kafka system, or the Yahoo Message Bus service used for asynchronous replication in the PNUTS distributed data storage system, allow a large number of topics, and use mul-

Telegram: @uni_k

www.konkur.in

tiple servers to handle messages to different topics in parallel. Thus, asynchronous replication can be made scalable.

Fault tolerance is an issue with asynchronous propagation of updates. If a primary node fails, a new node must take over as primary; this can be done either using an election algorithm, as we saw earlier or by having a master node (which is itself chosen by election) decide which node takes over the job of a failed primary node.

Consider what happens if a primary copy records an update but fails before the update is sent to the replicas. The new primary node has no way of finding out what was the last update committed at the primary copy. It can either wait for the primary to recover, which is unacceptable, or it can proceed without knowing what updates were committed just before failure. In the latter case, there is a risk that a transaction on the new primary may read an old value of a data item or perform an update that conflicts with an earlier update on the old primary.

To reduce the chance of such problems, some systems replicate the log records of the primary node to a backup node and allow the transaction to commit at the primary only after the log record has been successfully replicated at the backup node; if the primary node fails, the backup node takes over as the primary. Recall that this is the two-safe protocol from Section 19.7. This protocol is resilient to failure of one node, but not to the failure of two nodes.

If an application is built on top of a storage system using asynchronous replication, applications may potentially see some anomalous behaviors such as a read not seeing the effect of an earlier write done by the same application, or a later read seeing an earlier version of a data item than an earlier read, if different reads and writes are sent to different replicas. While such anomalies cannot be completely prevented in the event of failures, they can be avoided during normal operation by taking some precautions. For example, if read and write requests for a data item from a particular node are always sent to the same replica, the application will see any writes it has performed, and if two reads are performed on the same data item, the later read will see a version at least as new as the earlier read. This property is guaranteed if a primary replica is used to perform all actions on a data item.

23.6.3 Asynchronous View Maintenance

Indices and materialized views are forms of data derived from underlying data, and can thus be viewed as forms of replicated data. Just like replicas, indices and materialized views could be updated (maintained) as part of each transaction that updates the underlying data; doing so would ensure consistency of the derived data with the underlying data.

However, many systems prefer to perform index and view maintenance in an asynchronous manner, to reduce the overhead on transactions that update the underlying data. As a result, the indices and materialized views could be out of date. Any transac-

Telegram: @uni_k

www.konkur.in

tion that uses such indices or materialized views must be aware that these structures may be out of date.

We now consider how to maintain indices and materialized views in the face of concurrent updates.

- The first requirement for view maintenance is for the subsystem that performs maintenance to receive information about updates to the underlying data in such a way that each update is delivered exactly once, despite failures.

  Publish-subscribe systems are a good match for the first requirement above. All updates to any underlying relation are published to the pub-sub system with the relation name as the topic; the view maintenance subsystem subscribes to the topics corresponding to its underlying relations and received all relevant updates. As we saw in Section 22.8.1, we can have topics corresponding to each tablet of a stored relation. For a nonmaterialized intermediate relation that is partitioned, we can have a topic corresponding to each partition.

- The second requirement is for the subsystem to update the derived data in such a way that the derived data will be consistent with the underlying data, despite concurrent updates to the underlying data.

  Since the underlying data may receive further updates as an earlier update is being processed, no asynchronous view maintenance technique can guarantee that the view state is consistent with the state of the underlying data at all times. However, the consistency requirement can be formalized as follows: if there are no updates to the underlying data for a sufficient amount of time, asynchronous maintenance must ensure that the derived data is consistent with the underlying data; such a requirement is known as an **eventual consistency** requirement.

  The technique for parallel maintenance of materialized views which we saw in Section 22.7.5 uses the exchange operator model to send updates to nodes and allows view maintenance to be done locally. Techniques designed for view maintenance in a centralized setting can be used at each node, on locally materialized input data. Recall from Section 16.5.1 that view maintenance may be deferred, that is, it may be done after the transaction commits. Techniques for deferred view maintenance in a centralized setting already need to deal with concurrent updates; such techniques can be used locally at each node.

- A third requirement is for reads to get a consistent view of data. In general, a query that reads data from multiple nodes may not observe the updates of a transaction T on node N_1, but may see the updates that T performed on node N_2, thus seeing a transactionally inconsistent view of data. Systems that use asynchronous replication typically do not support transactionally consistent views of the database.

  Further, scans of the database may not see an operation-consistent view of the database. (Recall the notion of operation consistency from Section 18.9, which requires that any operation should not see a database state that reflects only some of the updates of another operation. In Section 18.9 we saw an example of a scan

Telegram: @uni_k

www.konkur.in

using an index that could see two versions, or neither version, of a record updated by a concurrent transaction, if the relation scan does not follow two-phase locking. A similar problem occurs with asynchronous propagation of updates, even if both the relation scan and the update transaction follow two-phase locking.

For example, consider a relation $r(A, B, C)$, with primary key A, which is partitioned on attribute B. Now consider a query that is scanning the relation r. Suppose there is a concurrent update to a tuple $t_1 \in r$, which updates attribute $t_1.B$ from v_1 to v_2. Such an update requires deletion of the old tuple from the partition corresponding to value v_1, and insertion of the new tuple in the partition corresponding to v_2. These updates are propagated asynchronously.

Now, the scan of r could possibly scan the node corresponding to v_1 after the old tuple is deleted there but visit the node corresponding to v_2 before the asynchronous propagation inserts the updated tuple in that node. Then, the scan would completely miss the tuple, even though it should have seen either the old value or the new value of t_1. Further, the scan could visit the node corresponding to v_1 before the delete is propagated to that node, and the node corresponding to v_2 after the insert is propagated to that node, and thereby see two versions of t_1, one from before the update and one from after the update. Neither case would be possible with two-phase locking, if updates are propagated synchronously to all copies.

If a multiversion concurrency control technique is used, where data items have timestamps, snapshot reads are a good way to get a consistent scan of a relation; the snapshot timestamp should be set a sufficiently old value that all updates as of that timestamp have reached all replicas.

23.6.4 Detecting Inconsistent Updates

Many applications developed for such high availability are designed to continue functioning locally even when the node running the application is disconnected from the other nodes.

As an example, when data are replicated, and the network gets partitioned, if a system chooses to trade off consistency to get availability, updates may be done concurrently at multiple replicas. Such conflicting updates need to be detected and resolved. When a connection is re-established, the application needs to communicate with a storage system to send any updates done locally and fetch updates performed elsewhere. There is a potential for conflicting updates from different nodes. For example, node N_1 may update a locally cached copy of a data item while it is disconnected; concurrently another node may have updated the data item on the storage system, or may have updated its own local copy of the data item. Such conflicting updates must be detected, and resolved.

As another example, consider an application on a mobile device that supports offline updates (i.e., permits updates even if the mobile device is not connected to the

Telegram: @uni_k

www.konkur.in

network). To give the user a seamless usage experience, such applications perform the updates on a locally cached copy, and then apply the update to the data store when the device goes back online. If the same data item may be updated from multiple devices, the problem of conflicting updates arises here, too. The schemes described below can be used in this context too, with nodes understood to also refer to mobile devices.

A mechanism for detecting conflicting updates is described in this section. How to resolve conflicting updates once they are detected is application dependent, and there is no general technique for doing so. However, some commonly used approaches are discussed in Section 23.6.5.

For data items updated by only one node, it is a simple matter to propagate the updates when the node gets reconnected to the storage system. If the node only caches read-only copies of data that may be updated by other nodes, the cached data may become inconsistent. When the node gets reconnected, it can be sent **invalidation reports** that inform it of out-of-date cache entries.

However, if updates can occur at more than one node, detecting conflicting updates is more difficult. Schemes based on **version numbering** allow updates of shared data from multiple nodes. These schemes do not guarantee that the updates will be consistent. Rather, they guarantee that, if two nodes independently update the same version of a data item, the clash will be detected eventually, when the nodes exchange information either directly or through a common node.

The **version-vector scheme** detects inconsistencies when replicas of a data item are independently updated. This scheme allows copies of a data item to be stored at multiple nodes.

The basic idea is for each node i to store, with its copy of each data item d, a **version vector**—that is, a set of version numbers $\{V[j]\}$, with one entry for each other node j on which the data item could potentially be updated. When a node i updates a data item d, it increments the version number $V[i]$ by one.

For example, suppose a data item is replicated at nodes N_1, N_2 and N_3. If the item is initially created at N_1, the version vector could be $[1, 0, 0]$. If it is then replicated at N_2, and then updated at node N_2, the resultant version vector would be $[1, 1, 0]$. Suppose now that this version of the data item is replicated to N_3, and then both N_2 and N_3 concurrently update the data item. Then, the version vector of the data item at N_2 would be $[1, 2, 0]$, while the version vector at N_3 would be $[1, 1, 1]$.

Whenever two nodes i and j connect with each other, they exchange updated data items, so that both obtain new versions of the data items. However, before exchanging data items, the nodes have to discover whether the copies are consistent:

1. If the version vectors V_i and V_j of the copy of the data item at nodes i and j are the same—that is, for each k, $V_i[k] = V_j[k]$—then the copies of data item d are identical.
2. If, for each k, $V_i[k] \leq V_j[k]$ and the version vectors are not identical, then the copy of data item d at node i is older than the one at node j. That is, the copy of

Telegram: @uni_k

www.konkur.in

data item d at node j was obtained by one or more modifications of the copy of the data item at node i. Node i replaces its copy of d, as well as its copy of the version vector for d, with the copies from node j.

In our example above, if N_1 had the vector $[1, 0, 0]$ for a data item, while N_2 had the vector $[1, 1, 0]$, then the version at N_2 is newer than the version at N_1.

3. If there are a pair of values k and m such that $V_i[k] < V_j[k]$ and $V_i[m] > V_j[m]$, then the copies are *inconsistent*; that is, the copy of d at i contains updates performed by node k that have not been propagated to node j, and, similarly, the copy of d at j contains updates performed by node m that have not been propagated to node i. Then, the copies of d are inconsistent, since two or more updates have been performed on d independently.

   In our example, after the concurrent updates at N_2 and N_3, the two version vectors show the updates are inconsistent. Let V_2 and V_3 denote the version vectors at N_2 and N_3. Then $V_2[2] = 2$ while $V_3[2] = 1$, whereas $V_2[3] = 0$, while $V_3[3] = 1$.

   Manual intervention may be required to merge the updates. After merging the updates (perhaps manually), the version vectors are merged, by setting $V[k]$ to the maximum of $V_i[k]$ and $V_j[k]$ for each k. The node l that performs the write then increments $V[l]$ by 1 and then writes the data item and its version vector V.

The version-vector scheme was initially designed to deal with failures in distributed file systems. The scheme gained importance because mobile devices often store copies of data that are also present on server systems. The scheme is also widely used in distributed storage systems that allow updates to happen even if a node is not in a majority partition.

The version-vector scheme cannot solve the problem of how to reconcile inconsistent copies of data detected by the scheme. We discuss reconciliation in Section 23.6.5.

The version-vector scheme works well for detecting inconsistent updates to a single data item. However, if a storage system has a very large number of replicated items, finding which items have been inconsistently updated can be quite expensive if done naively. In Section 23.6.6 we study a data structure called a Merkle tree that can efficiently detect differences between sets of data items.

23.6.5 Resolving Conflicting Updates

Detection of conflicting updates may happen when a read operation fetches copies of a data item from multiple replicas or when the system executes a background process that compares data item versions.

At that point, conflicting updates on the same data item need to be **resolved**, to create a single common version. Resolution of conflicting updates is also referred to as **reconciliation**.

Telegram: @uni_k

www.konkur.in

There is no technique for resolution so that can be used across all applications. We discuss some techniques that have been used in several commonly used applications.

Many applications can perform reconciliation automatically by executing on each node all update operations that had been performed on other nodes during a period of disconnection. This solution requires that the system keep track of operations, for example, adding an item to a shopping cart, or deleting an item from a shopping cart. This solution works if operations **commute**—that is, they generate the same result, regardless of the order in which they are executed. The addition of items to a shopping cart clearly commutes. Deletions do not commute with additions in general, which should be clear if you consider what happens if an addition of an item is exchanged with a delete of the same item. However, as long as deletion always operates only on items already present in the cart, this problem does not arise.

As another example, many banks allow customers to withdraw money from an ATM even if it is temporarily disconnected from the bank network. When the ATM gets reconnected, the withdrawal operation is applied to the account. Again, if there are multiple withdrawals, they may get merged in an order different from the order in which they happened in the real world, but the end result (balance) is the same. Note that since the operation already took place in the physical world, it cannot be rejected because of a negative balance; the fact that an account has a negative balance has to be dealt with separately.

There are other application-specific solutions for resolving conflicting updates. In the worst case, however, a system may need to alert humans to the conflicting updates, and let the humans decide how to resolve the conflict.

Dealing with such inconsistency automatically, and assisting users in resolving inconsistencies that cannot be handled automatically, remains an area of research.

23.6.6 Detecting Differences Between Collections Using Merkle Tree

The **Merkle tree** (also known as **hash tree**) is a data structure that allows efficient detection of differences between sets of data items that may be stored at different replicas. (To avoid confusion between tree nodes and system nodes, we shall refer to the latter as replicas in this section.)

Detecting items that have inconsistent values across replicas due to weak consistency is merely one of motivations for Merkle trees. Another motivation is performing sanity checks of replicas that are synchronously updated, and should be consistent, but may be inconsistent due to bugs or other failures. We consider below a binary version of the Merkle tree.

We assume that each data item has a key and a value; in case we are considering collections that do not have an explicit key, the data item value itself can be used as a key.

Each data item key k_i is hashed by a function $h_1()$ to get a hash value with n bits, where n is chosen such that 2^n is within a small factor of the number of data items. Each data item value v_i is hashed by another function h () to get a hash value (which

Telegram: @uni_k

www.konkur.in

is typically much longer than n bits). Finally, we assume a hash function $h_3()$ which takes as input a collection of hash values and returns a hash value computed from the collection (this hash function must be computed in a way that does not depend on the input order of the hash values, which can be done, for example, by sorting the collection before computing the hash function).

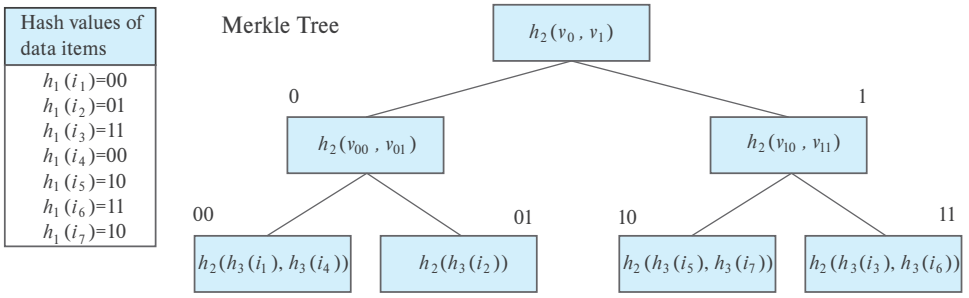
Each node of a Merkle tree has associated with it an identifier and stores a hash value. Each leaf of the tree can be identified by an n-bit binary number. For a given leaf identified by number k, consider the set of all data items i whose key k_i is such that $h_1(k_i) = k$. Then, the hash value v_k stored at leaf k is computed by applying $h_2()$ on each of the data item values v_i, and then applying $h_3()$ on the resultant collection of hash values. The system also maintains an index that can retrieve all the data items with a given hash value computed by function $h_2()$.

Figure 23.8 shows an example of a Merkle tree on 8 data items. The hash value of these data items on h_1 are shown on the left. Note that if for an item i_j, $h_1(i_j) = k$, then the data item i_j is associated with the leaf with identifier k.

Each internal node of the Merkle tree is identified by a hash value that is j bits long if the node is at depth j; leaves are at depth n, and the root at depth 0. The internal node identified by a number k has as children nodes identified by $2k$ and $2k + 1$. The hash value stored v_k at node k is computed by applying $h_3()$ to the hash value stored at nodes $2k$ and $2k + 1$.

Now, suppose this Merkle tree is constructed on the data at two replicas (the replicas may be whole database replicas, or replicas of a partition of the database). If all items at the two replicas are identical, the stored hash values at the root nodes will also be identical.

As long as $h_2()$ computes a long enough hash value, and is suitably chosen, it is very unlikely that $h_2(v_1) = h_2(v_2)$ if $v_1 \neq v_2$, and similarly for $h_3()$. The SHA1 hash function with a 160-bit hash value is an example of a hash function that satisfies this



Node identifier shown above node, and has value shown inside node,
v_i denotes stored hash value in node i

Figure 23.8 Example of Merkle tree.

Telegram: @uni_k

www.konkur.in

requirement. Thus, we can assume that if two nodes have the same stored hash values, all the data items under the two nodes are identical.

If, in fact, there is a difference in the value of any items at the two replicas, or if an item is present at one replica but not at the other, the stored hash values at the root will be different, with high probability.

Then, the stored hash values at each of the children are compared with the hash values at the corresponding child in the other tree. Search traverses down each child whose hash value differs, until a leaf is reached. The traversal is done in parallel on both trees and requires communication to send tree node contents from one replica to the other.

At the leaf, if the hash values differ, the list of data item keys associated with the leaves, and the corresponding data item values are compared across the two trees, to find data items whose values differ as well as data items that are present in one of the trees but not in the other.

One such traversal takes time at most logarithmic in the number of leaf nodes of the tree; since the number of leaf nodes is chosen to be close to the number of data items, the traversal time is also logarithmic in the number of data items. This cost is paid at most once for each data item that differs between the two replicas. Furthermore, a path to a leaf is traversed only if there is, in fact, a difference at the leaf.

Thus, the overall cost for finding differences between two (potentially very large) sets is $O(m \log_2 N)$, where m is the number of data items that differ and N is the total number of data items. Wider trees can be used to reduce the number of nodes encountered in a traversal, which would be $\log_K N$ if each node has K children, at the cost of more data being transferred for each node. Wider trees are preferred if network latency is high compared to the network bandwidth.

Merkle trees have many applications; they can be used to find the difference in contents of two databases that are almost identical without transferring large amounts of data. Such inconsistencies can occur due to the use of protocols that only guarantee weak consistency. They could also occur because of message or network failures that result in differences in replicas, even if consensus or other protocols that guarantee consistent reads are used.

The original use of Merkle trees was for **verification of the contents** of a collection that may have potentially been corrupted by malicious users. Here, the Merkle tree leaf nodes must store the hash values of all data items that map to it, or a tree variant that only stores one data item at a leaf may be used. Further, the stored hash value at the root is digitally signed, meaning its contents cannot be modified by a malicious user who does not have the private key used for signing the value.

To check an entire relation, the hash values can be recomputed from the leaves upwards, and the recomputed hash value at the root can be compared with the digitally signed hash value stored at the root.

To check consistency of a single data item, its hash value is recomputed; and then so is the hash value for its leaf node n_i, using existing hash values for other data items

Telegram: @uni_k

www.konkur.in

that hash to the same leaf. Next consider the parent node n_j of node n_i in the tree. The hash value of n_j is computed using the recomputed hash value of n_i and the already stored hash values of other children of n_j. This process is continued upward until the root of the tree. If the recomputed hash value at the root matches the signed hash value stored with the root, the contents of the data item can be determined to be uncorrupted.

The above technique works for detecting corruption since with suitably chosen hash functions, it is very hard for a malicious user to create replacement values for data items in a way that the recomputed hash value is identical to the signed hash value stored at the root.

23.7 Coordinator Selection

Several of the algorithms that we have presented require the use of a coordinator. If the coordinator fails because of a failure of the node at which it resides, the system can continue execution by restarting a new coordinator on another node. One way to continue execution is by maintaining a backup to the coordinator that is ready to assume responsibility if the coordinator fails. Another way is to "elect" a coordinator from among the nodes that are alive. We outline these options in this section. We then briefly describe fault-tolerant distributed services that have been developed to help developers of distributed applications perform these tasks.

23.7.1 Backup Coordinator

A **backup coordinator** is a node that, in addition to other tasks, maintains enough information locally to allow it to assume the role of coordinator with minimal disruption to the distributed system. All messages directed to the coordinator are received by both the coordinator and its backup. The backup coordinator executes the same algorithms and maintains the same internal state information (such as, for a concurrency coordinator, the lock table) as does the actual coordinator. The only difference in function between the coordinator and its backup is that the backup does not take any action that affects other nodes. Such actions are left to the actual coordinator.

In the event that the backup coordinator detects the failure of the actual coordinator, it assumes the role of coordinator. Since the backup has all the information available to it that the failed coordinator had, processing can continue without interruption.

The prime advantage of the backup approach is the ability to continue processing immediately. If a backup were not ready to assume the coordinator's responsibility, a newly appointed coordinator would have to seek information from all nodes in the system so that it could execute the coordination tasks. Frequently, the only source of some of the requisite information is the failed coordinator. In this case, it may be necessary to abort several (or all) active transactions and to restart them under the control of the new coordinator.

Telegram: @uni_k

www.konkur.in

Thus, the backup-coordinator approach avoids a substantial amount of delay while the distributed system recovers from a coordinator failure. The disadvantage is the overhead of duplicate execution of the coordinator's tasks. Furthermore, a coordinator and its backup need to communicate regularly to ensure that their activities are synchronized.

In short, the backup-coordinator approach incurs overhead during normal processing to allow fast recovery from a coordinator failure.

23.7.2 Election of Coordinator

In the absence of a designated backup coordinator, or in order to handle multiple failures, a new coordinator may be chosen dynamically by nodes that are live.

One possible approach is to have a designated node choose a new coordinator, when the current coordinator has failed. However, this raises the question of what to do if the node that chooses replacement coordinators itself fails.

If we have a fault-tolerant lock manager, a very effective way of choosing a new coordinator for a task is to use *lock leases*. The current coordinator has a lock lease on a data item associated with the task. If the coordinator fails, the lease will expire. If a participant determines that the coordinator may have failed, it attempts to get a lock lease for the task. Note that multiple participants may attempt to get a lease, but the lock manager ensures that only one of them can get the lease. The participant that gets the lease becomes the new coordinator. As discussed in Section 23.3.3, this ensures that only one node that can be the coordinator at a given time. Lock leases are widely used to ensure that a single node gets chosen as coordinator. However, observe that there is an underlying assumption of a fault-tolerant lock manager.

A participant determines that the coordinator may have failed if it is unable to communicate with the coordinator. Participants send periodic **heart-beat** messages to the coordinator and wait for an acknowledgment; if the acknowledgment is not received within a certain time, the coordinator is assumed to have failed.

Note that the participant cannot definitively distinguish a situation where the coordinator is dead from a situation where the network link between the node and the coordinator is cut. Thus, the system should be able to work correctly even if the current coordinator is alive, but another participant determines that the coordinator is dead. Lock leases ensure that at most one node can be the coordinator at any time; once a coordinator dies, another node can become the coordinator. However, lock leases work only if a fault-tolerant lock manager is available.

This raises the question of how to implement such a lock manager. We return later, in Section 23.8.4, to the question of how to implement a fault-tolerant lock manager. But it turns out that to do so efficiently, we need to have a coordinator. And, lock leases cannot be used to choose the coordinator for the lock manager! The problem of how to choose a coordinator without depending on a lock manager is solved by **election algorithms**, which enable the participating nodes to choose a new coordinator in a decentralized manner.

Telegram: @uni_k

www.konkur.in

Suppose the goal is to elect a coordinator just once. Then, each node that wishes to become the coordinator proposes itself as a candidate to all the other nodes; such nodes are called *proposers*. The participating nodes then vote on which node among the candidates is to be chosen. If a majority of the participating nodes (called *acceptors*) vote for a particular candidate, it is chosen. A subset of nodes called *learners* ask the acceptor nodes for their vote and determine if a majority have voted for a particular candidate.

The problem with the above idea is that if there are multiple candidates, none of them may get a majority of votes. The question is what to do in such a situation. There are at least two approaches that have been proposed:

- Nodes are given unique numbers; if more than one candidate proposes itself, acceptors choose the highest-numbered candidate. Even then votes may be split with no majority decision, due to delayed or missing messages; in such a case, the election is run again. But if a node N_1 that was a candidate finds that a higher-numbered node N_2 has proposed itself as a candidate, then N_1 withdraws from the next round of the election. The highest-numbered candidate will win the election. The **bully algorithm** for election is based on this idea.

  There are some subtle details due to the possibility that the highest-numbered candidate in one round may fail during a subsequent round, leading to there being no candidates at all! If a proposer observes that no coordinator was selected in a round where it withdrew itself as a candidate, it proposes itself as a candidate again in the next round.

  Note also that the election has multiple rounds; each round has a number, and a candidate attaches a round number with the proposal. The round number is chosen to be the maximum round that it has seen, plus 1. A node can give a vote to only one candidate in a particular round, but it may change its vote in the next round.

- The second approach is based on *randomized retry*, which works as follows: If there is no majority decision in a particular round, all participants wait for a randomly chosen amount of time; if by that time a coordinator has been chosen by a majority of nodes, it is accepted as a coordinator. Otherwise, after the timeout, the node proposes itself as a candidate. As long as the timeouts are chosen properly (large enough compared to network latency) with high likelihood only one node proposes itself at a particular time and will get votes from a majority of nodes in a particular round.

  If no candidate gets a majority vote in a round, the process is repeated. With very high probability, after a few rounds, one of the candidates gets a majority and is thus chosen as coordinator.

  The randomized-retry approach was popularized by the Raft consensus algorithm, and it is easier to reason about it and show not just correctness, but also bounds on the expected time for an election round to succeed in choosing a coordinator, as compared to the node- numbering-based approach.

Telegram: @uni_k

www.konkur.in

Note that the above description assumed that choosing a coordinator is a one-time activity. However, the chosen coordinator may fail, requiring a fresh election algorithm. The notion of a **term** is used to deal with this situation. As mentioned above, each time a node proposes itself as a coordinator, it associates the proposal with a round number, which is 1 more than the highest round number it has seen earlier, after ensuring that in the previous round no coordinator was chosen, or the chosen coordinator has subsequently failed. The round number is henceforth referred to as a **term**. When the election succeeds, the chosen coordinator is the coordinator for the corresponding term. If the election fails, the corresponding term does not have any coordinator chosen, but the election should succeed in a subsequent term.

Note also that there are subtle issues that arise since a node n may be disconnected from the network for a while, and it may get reconnected without ever realizing that it was disconnected. In the interim, the coordinator may have changed. In particular, if the node n was the coordinator, it may continue to think it is the coordinator, and some other node, say N_1, which was also disconnected may think that n is still coordinator. However, if a coordinator was successfully elected, the majority of the nodes agree that some other node, say N_2, is the coordinator.

In general, it is possible for more than one node to think that is the coordinator at the same time, although at most one of them can have the majority vote at that point in time.

To avoid this problem, each coordinator can be given a lease for a specified period. The coordinator can extend the lease by requesting an extension from other nodes and getting confirmation from a majority of the nodes. But if the coordinator is disconnected from a majority of the nodes, it cannot renew its lease, and the lease expires. A node can vote for a new coordinator only if the last lease time that it confirmed to the earlier coordinator has expired. Since a new coordinator needs a majority vote, it cannot get the vote until the lease time of the previous coordinator has expired.

However, even if leases are used to ensure that two nodes cannot be coordinators at the same time, delayed messages can result in a node getting a message from an old coordinator after a new one has been elected.

To deal with this problem, the current term of the sender is included with each message exchanged in the system. Note that when a node n is elected as coordinator, it has an associated term t; participant nodes that learn that n is the coordinator are aware of the current term t. A node may receive a message with an old term either because an old coordinator did not realize it has been replaced or because of message delivery delay; the latter problem can occur even if leases or other mechanisms ensure that only one node can be the coordinator at a time. In either case, a node that receives a **stale message**, that is, one with a term older than the current term of the node, it can ignore the message. If a node receives a message with a higher number, it is behind the rest of the system, and it needs to find out the current term and coordinator by contacting other nodes.

Some protocols do not require the coordinator to store any state information; in such cases, the new coordinator can take over without any further actions. However,

Telegram: @uni_k

www.konkur.in

other protocols require coordinators to retain state information. In such cases, the new coordinator has to reconstruct the state information from persistent data and recovery logs created by the previous coordinator. Such logs, in turn, need to be replicated to multiple nodes so that the loss of a node does not result in the loss of access to the recovery data. We shall see how to ensure availability by means of data replication in subsequent sections.

23.7.2.1 Distributed Coordination Services

There are a very large number of distributed applications that are in daily use today. Instead of each one having to implement its own mechanism for electing coordinators (among other tasks), it makes sense to develop a fault-tolerant coordination service that can be used by multiple distributed applications.

The **ZooKeeper** service is one such very widely used fault-tolerant distributed coordination service. The **Chubby** service developed earlier at Google is another such service, which is widely used for applications developed by Google. These services internally use consensus protocols to implement fault tolerance; we study consensus protocols in Section 23.8.

These services provide a file-system-like API, which supports the following features, among others:

- *Store (small amounts of) data* in files, with a hierarchical namespace. A typical use for such storage is to store configuration information that can be used to start up a distributed application, or for new nodes to join a distributed application by finding out which node is currently the coordinator.
- *Create and delete files*, which can be used to implement locking. For example, to get a lock, a process can attempt to create a file with a name corresponding to the lock. If another process has already created the file, the coordination service will return an error, so the process knows it could not get the lock.

  For example, a node that acts as a master for a tablet in a key-value store would get a lock on a file whose name is the identifier of the tablet. This ensures that two nodes cannot be masters for the tablet at the same time.

  If an overall application master detects that a tablet master has died, it could release the lock. If the service supports lock leases, this could happen automatically, if the tablet master does not renew its lease.
- *Watch for changes on a file*, which can be used by a process to check if a lock has been released, or to be informed about other changes in the system that require action by the process.

23.8 Consensus in Distributed Systems

In this section we first describe the consensus problem in a distributed system, that is, how a set of nodes agree on a decision in a fault-tolerant

Telegram: @uni_k

www.konkur.in

a key building block for protocols that update replicated data in a fault-tolerant manner. We outline two consensus protocols, Paxos and Raft. We then describe replicated state machines, which can be used to make services, such as data storage systems and lock managers, fault tolerant. We end the section by describing how consensus can be used to make two-phase commit nonblocking.

23.8.1 Problem Overview

Software systems need to make decisions, such as the coordinator's decision on whether to commit or abort a transaction when using 2PC, or a decision on which node is to act as coordinator, in case a current coordinator fails.

If the decision is made by a single node, such as the commit/abort decision made by a coordinator node in 2PC, the system may block in case the node fails, since other nodes have no way of determining what decision was made. Thus, to ensure fault tolerance, multiple nodes must participate in the decision protocol; even if some of these nodes fail, the protocol must be able to reach a decision. A single node may make a proposal for a decision, but it must involve the other nodes to reach a decision in a fault-tolerant manner.

The most basic form of the **distributed consensus problem** is thus as follows: a set of n nodes (referred to as *participants*) need to agree on a decision by executing a protocol such that:

- All participants must "learn" the same value for the decision even if some nodes fail during the execution of the protocol, or messages are lost, or there are network partitions.
- The protocol should not block, and must terminate, as long as some majority of the nodes participating remain alive and can communicate with each other.

Any real system cannot just make a single decision once, but needs to make a series of decisions. A good abstract of the process of making multiple consensus decisions is to treat each decision as *adding a record to a log*. Each node has a copy of the log, and records are appended to the log at each node. There can potentially be conflicts on what record is added at what point in a log. The **multiple consensus protocol** viewed from this perspective needs to ensure that the log is uniquely defined.

Most consensus protocols allow temporary divergence of logs across nodes while the protocol is being executed; that is, the same log position at different nodes may have different records, and the end of the log may be different at different nodes. Shared-log consensus protocols keep track of an index into the log such that any entry before that index has definitely been agreed upon. Any entries after that index may be in the process of being agreed upon, or may be entries from failed attempts at consensus. However, the protocols subsequently bring the inconsistent parts of the log logs back in synchronization. To do this, log records at some nodes may be deleted after being inserted; such log records are viewed as not yet committed and cannot be used to make

Telegram: @uni_k

www.konkur.in

decisions. Only log records in the prefix of the log that are in the committed prefix may be used to make decisions.

Several protocols have been proposed for distributed consensus. Of these, the Paxos family of protocols is one of the most popular, and it has been implemented in many systems. While the basic Paxos protocol is intuitively easy to understand at a high level, there are a number of details in its implementation that are rather complicated, and particularly so in the multiple consensus version. To address this issue, the Raft consensus protocol was developed, with ease of understanding and implementation being key goals, and it has been adopted by many systems. We outline the intuition behind these protocols in this section.

A key idea behind distributed consensus protocols is the idea of *voting* to make a decision; a particular decision succeeds only if a majority of the participating nodes vote for it. Note that if two or more different values are proposed for a particular decision, at most one of them can be voted for by a majority; thus, it is not possible for two different values to be chosen. Even if some nodes fail, if a majority of the participants vote for a value, it gets chosen, thus making the voting fault tolerant as long as a majority of the participants are up and connected to each other. There is, however, a risk that votes may get split between the proposed values, and some nodes may not vote if they fail; as a result, no value may be decided on. In such a case the voting procedure has to be executed again.

While the above intuition is easy enough to understand, there are many details that make the protocols nontrivial. We study some of these issues in the following sections.

We note that although we study some of the features of the Paxos and Raft consensus protocols, we omit a number of details that are needed for correct operation to keep our description concise.

We also note that a number of other consensus protocols have been proposed, and some of them are widely used, such as the Zab protocol which is part of the ZooKeeper distributed coordination service.

23.8.2 The Paxos Consensus Protocol

The basic Paxos protocol for making a single decision has the following participants.

1. One or more nodes that can propose a value for the decision; such nodes are called **proposers**.
2. One or more nodes that act as **acceptors**. An acceptor may get proposals with different values from different proposers and must choose (vote for) only one of the values.

   Note that failure of an acceptor does not cause a problem, as long as a majority of the acceptors are live and reachable. Failure or disconnection of a majority would block the consensus protocol.

Telegram: @uni_k

www.konkur.in

3. A set of nodes, called **learners**, query the acceptors to find what value each acceptor voted for in a particular round. (Acceptors could also send the value they accepted to the learners, without waiting for a query from the learner.)

Note that the same node can play the roles of proposer, acceptor, and learner.

If a majority of the acceptors voted for a particular value, that value is the chosen (consensus) value for that decision. But there are two problems:

1. It is possible for votes to be split among multiple proposals, and no proposal is accepted by a majority of the acceptors.

   If any proposed value is to get a majority, at least some acceptors must change their decision. Thus, we must allow another round of decision making, where acceptors may choose a new value. This may need to be repeated as long as required until one value wins a majority vote.

2. Even if a majority of nodes do accept a value, it is possible that some of these nodes die or get disconnected after accepting a value, but before any learner finds out about their acceptance, and the remaining acceptors of that value do not constitute a majority.

   If this is treated as the failure of a round, and a different value is chosen in a subsequent round, we have a problem. In particular, a learner that learned about the earlier majority would conclude that a particular value was chosen, while another learner could conclude that a different value was chosen, which is not acceptable.

Note also that acceptors must log their decision so when they recover they know what decision they made earlier.

The first problem above, namely, split votes, does not affect correctness, but it affects performance. To avoid this problem, Paxos makes use of a coordinator node. Proposers send a proposal to the coordinator, which picks one of the proposed values and follows the preceding steps to get a majority vote. If proposals come from only one coordinator, there is no conflict, and the lone proposed value gets a majority vote (modulo network and node failures).

Note that if the coordinator dies or is unreachable, a new coordinator can be elected, using techniques we saw earlier in Section 23.7, and the new coordinator can then do the same job as the earlier coordinator. Coordinators have no local state, so the new one can take over without any recovery steps.

The second problem, namely, different values getting majorities in different rounds, is a serious problem and must be avoided by the consensus protocol. To do so, Paxos uses the following steps:

1. Each proposal in Paxos has a number; different proposals must have different numbers.

Telegram: @uni_k

www.konkur.in

2. In phase 1a of the protocol, a proposer sends a *prepare* message to acceptors, with its proposal number n.

3. In phase 1b of the protocol, an acceptor that receives a prepare message with number n checks if it has already responded to a message with a number higher than n. If so, it ignores the message. Otherwise, it remembers the number n and responds with the highest proposal number $m < n$ that it has already accepted, along with the corresponding value v; if it has not accepted any value earlier, it indicates so in its response. (Note that responding is different from accepting.)

4. In phase 2a, the proposer checks if it got a response from a majority of the acceptors. If it does, it chooses a value v as follows: If none of the acceptors has already accepted any value, the proposer may use whatever value it intended to propose. If at least one of the acceptors responded that it accepted a value v with some number m, the proposer chooses the value v that has the highest associated number m (note that m must be $< n$).

   The proposer now sends an *accept* request with the chosen value v and number n.

5. In phase 2b, when an acceptor gets an accept request with value v and number n, it checks if it has responded to a prepare message with number $n_1 > n$; if so it ignores the accept request. Otherwise, it accepts the proposed value v with number n.

The above protocol is quite clever, since it ensures the following: if a majority of acceptors accepted a value v (with any number n), then even if there are further proposals with number $n_1 > n$, the value proposed will be value v. Intuitively, the reason is that a value can be accepted with number n only if a majority of nodes respond to a prepare message with number n; let us call this set of acceptors P. Suppose a value v had been accepted earlier by a majority of nodes with number m; call this set of nodes A. Then A and P must have a node in common, and the common node will respond with value v and number m.

Note that some other proposal with a number $p > n$ may have been made earlier, but if it had been accepted by even one node, then a majority of nodes would have responded to the proposal with number p, and thus will not respond to the proposal with number n. Thus, if a proposal with value v is accepted by a majority of nodes, we can be sure that any further proposal will be for the already chosen value v.

Note that if a learner finds that no proposal was accepted by a majority of nodes, it can ask any proposer to issue a fresh proposal. If a value v had been accepted by a majority of nodes, it would be found and accepted again, and the learner would now learn about the value. If no value was accepted by a majority of nodes earlier, the new proposal could be accepted.

The above algorithm is for a single decision. Paxos has been extended to allow a series of decisions; the extended algorithm is called Multi-Paxos. Real implementations

Telegram: @uni_k

www.konkur.in

also need to deal with other issues, such as how to add a node to the set of acceptors, or to remove a node from the set of acceptors if it is down for a long time, without affecting the correctness of the protocol. References with more details about Paxos and Multi-Paxos may be found in the bibliographic notes for this chapter, available online.

23.8.3 The Raft Consensus Protocol

There are several consensus protocols whose goal is to maintain a log, to which records can be appended in a fault-tolerant manner. Each node participating in such a protocol has a replica of the log. Log-based protocols simplify the handling of multiple decisions. The Raft consensus protocol is an example of such a protocol, and it was designed to be (relatively) easy to understand.

A key goal of log-based protocols is to keep the log replicas in sync by presenting a logical view of appending records atomically to all copies of the log. In fact, atomically appending the same entry to all replicas is not possible, due to failures. Recall that failure modes may include a node being temporarily disconnected and missing some updates, without ever realizing it was disconnected. Further, a log append may be done at just a few nodes, and the append process may fail subsequently, leaving other replicas without the record. Thus, ensuring all copies of the log are identical at all times is impossible. Such protocols must ensure the following:

- Even if a log replica is temporarily inconsistent with another, the protocol will bring it back in sync eventually by deleting and replacing log records on some copies.
- A log entry will not be treated as committed until the algorithm guarantees that it will never be deleted.

Protocols such as Raft that are based on log replication can allow each node to run a "state machine," with log entries used as commands to the state machine; state machines are described in Section 23.8.4.

The Raft algorithm is based on having a coordinator, which is called a **leader** in Raft terminology. The other participating nodes are called **followers**. Since leaders may die and need to be replaced, time is divided into terms, which are identified by integers. Each term has a unique leader, although some terms may not have any associated leader. Later terms have higher identifiers than earlier terms.

Leaders are elected in Raft using the randomized-retry algorithm outlined in Section 23.7.2. Recall that the randomized-retry algorithm already incorporates the notion of a term. A node that votes for a leader does so for a specific term. Nodes keep track of the currentTerm based on messages from leaders or requests for votes.

Note that a leader N_1 may get temporarily disconnected, but get reconnected after other nodes find the leader cannot be reached, and elect a new leader N . Node N

Telegram: @uni_k

www.konkur.in

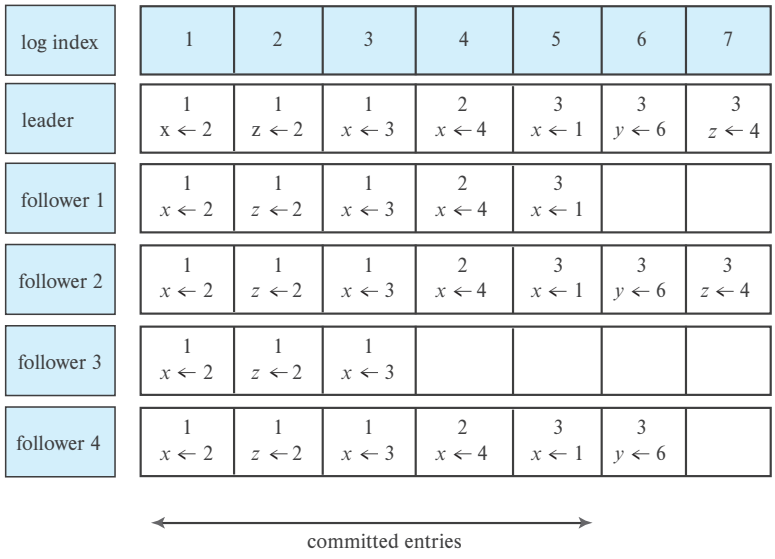


Figure 23.9 Example of Raft logs.

does not know that there is a new leader and may continue to execute the actions of a leader. The protocol should be robust to such situations.

Figure 23.9 shows an example of Raft logs at a leader and four followers. Note that the log index denotes the position of a particular record in a log. The number at the top of each log record is the term in which the log record was created, while the part below it shows the log entry, assumed here to record assignments to different variables.

Any node that wishes to append a record to the replicated log sends a log append request to the current leader. The leader adds its term as a field of the log records and appends the record to its log; it then sends an *AppendEntries* remote procedure call to the other nodes; the call contains several parameters, including these:

- *term*: the term of the current leader.
- *previousLogEntryPosition*: the position in the log of the preceding log entry.
- *previousLogEntryTerm*: the term associated with the preceding log entry.
- *logEntries*: an array of log records, allowing the call to append multiple log records at the same time.
- *leaderCommitIndex*: an index such that all log records at that index or earlier are committed. Recall that a log entry is not considered committed until a leader has confirmed that a majority of nodes have accepted that log entry. The leader keeps, in leaderCommitIndex, a position in the log such that all log records at that index and earlier are committed; this value is sent along with the AppendEntries call so that the nodes learn which log records have been committed.

Telegram: @uni_k

www.konkur.in

If a majority of the nodes respond to the call with a return value true, the leader can report successful log append (along with the position in the log) to the node that initiated the log append. We will shortly see what happens if a majority do not respond with true.

Each follower that receives an AppendEntries message does the following:

1. If the term in the message is less than currentTerm, then Return false.
2. If the log does not contain an entry at a previous log entry position, whose term matches the term in the message, then Return false.
3. If there is an existing entry at the log position that is different from the first log record in the AppendEntries message, the existing entry and all subsequent log entries are deleted.
4. Any log records in the logEntries parameter that are not already in the log are appended to the log.
5. The follower also keeps track of a local commitIndex to track which records are committed. If the leaderCommitIndex > commitIndex, set commitIndex = min(leaderCommitIndex, index of last entry in log).
6. Return true.

Note that the last step keeps track of the last committed log record. It is possible that the leader's log is ahead of the local log, so commitIndex cannot be blindly set to leaderCommitIndex, and it may need to be set to the local end of log if the leaderCommitIndex is ahead of the local end of log.

Figure 23.9 shows that different followers may have different log states, since some AppendEntries messages may not have reached those nodes. The part of the log up to entry 6 is present at a majority of nodes (namely, the leader, follower 2 and follower 4). On receipt of a true response to the AppendEntries call for the log record at position 6 from these followers, the leader can set leaderCommitIndex to 6.

It is possible for a node N_1 to be a leader in some term, and on temporary disconnection it may get replaced by a new leader N_2 in the next term. N_1 may not realize that there is a new leader for some time and may send appendEntry messages to other nodes. However, a majority of the other nodes will know about the new leader, and would have a term higher than that of N_1. Thus, these nodes would return false, and include their current term in the response. Node N_1 would then realize that there is a leader with a new term; it then switches from the role of leader to that of follower.

The protocol must deal with the fact that some nodes may have outdated logs. Note that in step 2 of the follower protocol, the follower returns false if its log is outdated. In such a case, the leader will retry an AppendEntries, sending it all log records from an even earlier point in the log. This may happen several times, until the leader sends log

Telegram: @uni_k

www.konkur.in

records from a point that is already in the follower log. At this point, the AppendEntries command would succeed.

A key remaining problem is to ensure that if a leader dies, and another one takes over, the log is brought to a consistent state. Note that the leader may have appended some log entries locally and replicated some of them to some other nodes, but the new leader may or may not have all these records. To deal with this situation, the Raft protocol includes steps to ensure the following:

1. The protocol ensures that any node elected as leader has all the committed log entries. To do so, any candidate must contact a majority of the nodes and send information about its log state when seeking a vote. A node will vote for a candidate in the election only if it finds that the candidate's log state is *at least as up-to-date* as its own; note that the definition of "at least as up-to-date" is a little complicated since it involves term identifiers in log records, and we omit details.

   Since the above check is done by a majority of the nodes that voted for the new leader, any committed entry would certainly be present in the log of the newly elected leader.

2. The protocol then forces all other nodes to replicate the leader's log.

   Note that the first step above does not actually find up to what log record is committed. Some of the log records at the new leader may not have been committed earlier, but may get committed when the new leader's log is replicated in this step.

   There is also a subtle detail in that the new leader cannot count the number of replicas with a particular record from an earlier term, and declare it committed if it is at a majority of the nodes. Intuitively, the problem is because of the definition of "at least as up-to-date" and the possibility that a leader may fail, recover, and be elected as leader again. We omit details, but note that the way this problem is solved is for the new leader to replicate a new log record in its current term; when that log record is determined to be at a majority of the replicas, it and all earlier log records can be declared to be committed.

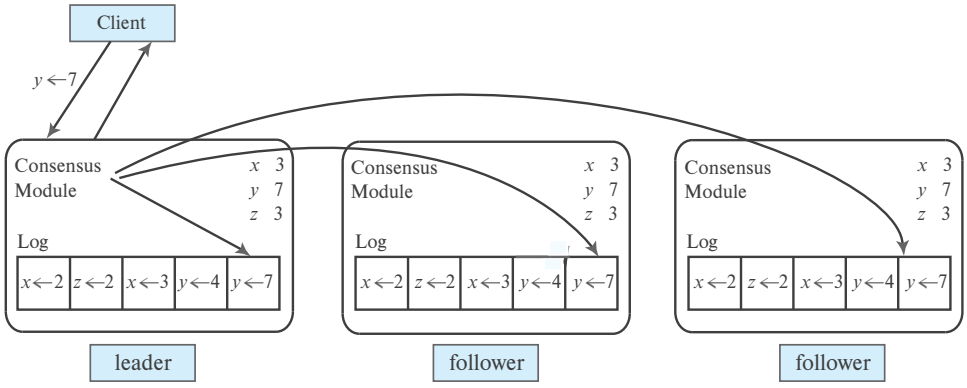
It should be clear that although the protocol, like Paxos, seems simple at a high level, there are many subtle details that need to be taken care of to ensure consistency even in the face of multiple failures and restarts. There are further details to be taken care of, including how to change the *cluster membership*, that is, the set of nodes that form the system, while the system is running (doing so carelessly can result in inconsistencies). Details of the above steps, including proofs of correctness, may be found in references in the bibliographic notes for this chapter, available online.

23.8.4 Fault-Tolerant Services Using Replicated State Machines

A key requirement in many systems is for a service to be made fault tolerant. A lock manager is an example of such a service, as is a key-value storage system.

Telegram: @uni_k

www.konkur.in



Leader declares log record committed after it is replicated at a majority of nodes. Update of state machine at each replica happens only after log record has been committed.

Figure 23.10 Replicated state machine.

A very powerful approach to making services fault tolerant is to model them as "state machines" and then use the idea of replicated state machines that we describe next.

A **state machine** receives inputs and has a stored state; it makes state transitions on each input and may output some results along with the state transition. A **replicated state machine** is a state machine that is replicated on multiple nodes to make it fault tolerant. Intuitively, even if one of the nodes fails, the state and output can be obtained from any of the nodes that are alive, provided all the state machines are in a consistent state. The key to ensuring that the state machine replicas are consistent is to (a) require the state machines to be deterministic, and (b) ensure that all replicas get exactly the same input in the same order.

To ensure that all replicas get exactly the same input in the same order, we just append the inputs to a replicated log, using, for example, techniques we saw earlier in Section 23.8.3. As soon as a log entry is determined to be committed, it can be given as input to the state machine, which can then process it.

Figure 23.10 depicts a replicated state machine based on a replicated log. When a client issues a command, such as $y \leftarrow 7$ in the figure, the command is sent to the leader, where the command is appended to the log. The leader then replicates the command to the logs at the followers. Once a majority have confirmed that the command has been replicated in their logs, the leader declares the command committed and applies the command to its state machine. It also informs the followers of the commit, and the followers then apply the command to their state machine.

In the example in Figure 23.10, the state machine merely records the value of the updated variable; but in general, the state machine may execute any other actions. The actions are, however, required to be deterministic, so all state machines are in exactly

Telegram: @uni_k

www.konkur.in

the same state when they have executed the same set of commands; the order of execution of commands will be the same since commands are executed in the log order.

Commands such as lock request must return a status to the caller. The status can be returned from any one of the replicas where the command is performed. Most implementations return the status from the leader node, since the request is sent to the leader, and the leader is also the first node to know when a log record has been committed (replicated to a majority of the nodes).

We now consider two applications that can be made fault-tolerant using the replicated state machine concept.

We first consider how to implement a **fault-tolerant lock manager**. A lock manager gets commands, namely, lock requests and releases, and maintains a state (lock table). It also gives output (lock grants or rollback requests on deadlock) on processing inputs (lock requests or releases). Lock managers can easily be coded to be deterministic, that is, given the same input, the state and output will be the same even if the code is executed again on a different node.

Thus, we can take a centralized implementation of a lock manager and run it on each node. Lock requests and releases are appended to a replicated log using, for example, the Raft protocol. Once a log entry is committed, the corresponding command (lock request or release) can be processed, in order, by the lock manager code at each replica. Even if some of the replicas fail, the other replicas can continue processing as long as a majority are up and connected.

Now consider the issue of implementing a **fault-tolerant key-value store**. A single-node storage system can be modeled as a state machine that supports `put()` and `get()` operations. The storage system is treated as a state machine, and the state machine is run on multiple nodes.

The `put()` operations are appended to the log using a consensus protocol and are processed when the consensus protocol declares the corresponding log records to be committed (i.e., replicated to a majority of the nodes).

If the consensus protocol uses leaders, `get()` operations need not be logged, and need to be executed only on the leader. To ensure that a `get()` operation sees the most recent `put()` on the same data item, all `put()` operations on the same data item that precede the `get()` operation in the log must be committed before the `get()` operation is processed. (If a consensus protocol does not use a leader, `get()` operations can also be logged and executed by at least one of the replicas which returns the value to the caller.)

Google's Spanner is an example of a system that uses the replicated state machine approach to creating a fault-tolerant implementation of a key-value storage system and a lock manager.

To ensure scalability, Spanner breaks up data into partitions, each of which has a subset of the data. Each partition has its data replicated across multiple nodes. Each node runs two state machines: one for the key-value storage system, and one for the lock manager. The set of replicas for a particular partition are called a *Paxos group*; one

Telegram: @uni_k

www.konkur.in

of the nodes in a Paxos group acts as the Paxos group leader. Lock manager operations, as well as key-value store operations for a particular partition, are initiated at the Paxos group leader for that partition. The operations are appended to a log, which is replicated to the other nodes in the Paxos group using the Paxos consensus protocol.[2] Requests are applied in order at each member of the Paxos group, once they are committed.

As an optimization, get() operations are not logged, and executed only at the leader as described earlier. As a further optimization, Spanner allows reads to run as of a particular point in time, allowing reads to be executed at any replica of the partition (in other words, any other member of the Paxos group) that is sufficiently up to date, based on the multiversion two-phase locking protocol described earlier in Section 23.5.1.

23.8.5 Two-Phase Commit Using Consensus

Given a consensus protocol implementation, we can use it to create a **non-blocking two-phase commit** implementation. The idea is simple: instead of a coordinator recording its commit or abort decision locally, it uses a consensus protocol to record its decision in a replicated log. Even if the coordinator subsequently fails, other participants in the consensus protocol know about the decision, so the blocking problem is avoided.

In case the coordinator fails before making a decision for a transaction, a new coordinator can first check the log to see if a decision was made earlier, and if not it can make a commit/abort decision and use the consensus protocol to record the decision.

For example, in the Spanner system developed by Google, a transaction may span multiple partitions. Two-phase commit is initiated by a client and coordinated by the Paxos group leader at one of the partitions where the transaction executed. All other partitions where an update was performed acts as a participant in the two-phase commit protocol. Prepare and commit messages are sent to the Paxos group leader node of each of the partitions; recall that two-phase commit participants as well as coordinators record decisions in their local logs. These decisions are recorded by each leader, using consensus involving all the other nodes in its Paxos group.

If a Paxos group member other than the leader dies, the leader can continue processing the two-phase commit steps, as long as a majority of the group nodes are up and connected. If a Paxos group leader fails, one of the other group members takes over as the group leader. Note that all the state information required to continue commit processing is available to the new leader. Log records written during commit processing are available since the log is replicated. Also, recall from Section 23.8.4 that Spanner makes the lock manager fault tolerant by using the replicated state machine concept. Thus, a consistent replica of the lock table is also available with the new leader. Thus, the two-phase commit steps of both the coordinator and the participants can continue to be executed even if some nodes fail.

[2]The Multi-Paxos version of Paxos is used, but we shall just refer

Telegram: @uni_k

www.konkur.in

23.9 Summary

- A distributed database system consists of a collection of sites or nodes, each of which maintains a local database system. Each node is able to process local transactions: those transactions that access data in only that single node. In addition, a node may participate in the execution of global transactions: those transactions that access data in several nodes. Transaction managers at each node manage access to local data, while the transaction coordinator coordinates execution of global transactions across multiple nodes.
- A distributed system may suffer from the same types of failure that can afflict a centralized system. There are, however, additional failures with which we need to deal in a distributed environment, including the failure of a node, the failure of a link, loss of a message, and network partition. Each of these problems needs to be considered in the design of a distributed recovery scheme.
- To ensure atomicity, all the nodes in which a transaction T executed must agree on the final outcome of the execution. T either commits at all nodes or aborts at all nodes. To ensure this property, the transaction coordinator of T must execute a commit protocol. The most widely used commit protocol is the two-phase commit protocol.
- The two-phase commit protocol may lead to blocking, the situation in which the fate of a transaction cannot be determined until a failed node (the coordinator) recovers. We can use distributed consensus protocols, or the three-phase commit protocol, to reduce the risk of blocking.
- Persistent messaging provides an alternative model for handling distributed transactions. The model breaks a single transaction into parts that are executed at different databases. Persistent messages (which are guaranteed to be delivered exactly once, regardless of failures), are sent to remote nodes to request actions to be taken there. While persistent messaging avoids the blocking problem, application developers have to write code to handle various types of failures.
- The various concurrency-control schemes used in a centralized system can be modified for use in a distributed environment. In the case of locking protocols, the only change that needs to be incorporated is in the way that the lock manager is implemented. Centralized lock managers are vulnerable to overloading and to failures. Deadlock detection in a distributed-lock-manager environment requires cooperation between multiple nodes, since there may be global deadlocks even when there are no local deadlocks.
- The timestamp ordering and validation based protocols can also be extended to work in a distributed setting. Timestamps used to order transactions need to be made globally unique.

Telegram: @uni_k

www.konkur.in

- Protocols for handling replicated data must ensure consistency of data. Linearizability is a key property that ensures that concurrent reads and writes to replicas of a single data item can be serialized.
- Protocols for handling replicated data include the primary copy, majority, biased, and quorum consensus protocols. These have different trade-offs in terms of cost and ability to work in the presence of failures.
- The majority protocol can be extended by using version numbers to permit transaction processing to proceed even in the presence of failures. While the protocol has a significant overhead, it works regardless of the type of failure. Less-expensive protocols are available to deal with node failures, but they assume network partitioning does not occur.
- To provide high availability, a distributed database must detect failures, reconfigure itself so that computation may continue, and recover when a processor or a link is repaired. The task is greatly complicated by the fact that it is hard to distinguish between network partitions and node failures.
- Globally consistent and unique timestamps are key to extending multiversion two-phase locking and snapshot isolation to a distributed setting.
- The CAP theorem indicates that one cannot have consistency and availability in the face of network partitions. Many systems tradeoff consistency to get higher availability. The goal then becomes eventual consistency, rather than ensuring consistency at all times. Detecting inconsistency of replicas can be done by using version vector schemes and Merkle trees.
- Many database systems support asynchronous replication, where updates are propagated to replicas outside the scope of the transaction that performed the update. Such facilities must be used with great care, since they may result in nonserializable executions.
- Some of the distributed algorithms require the use of a coordinator. To provide high availability, the system must maintain a backup copy that is ready to assume responsibility if the coordinator fails. Another approach is to choose the new coordinator after the coordinator has failed. The algorithms that determine which node should act as a coordinator are called election algorithms. Distributed coordination services such as ZooKeeper support coordinator selection in a fault-tolerant manner.
- Distributed consensus algorithms allow consistent updates of replicas, even in the presence of failures, without requiring the presence of a coordinator. Coordinators may still be used for efficiency, but failure of a coordinator does not affect correctness of the protocols. Paxos and Raft are widely used consensus protocols. Replicated state machines, which are implemented using consensus algorithms, can be used to build a variety of fault-tolerant services.

Telegram: @uni_k

www.konkur.in

Review Terms

- Distributed transactions
  - Local transactions
  - Global transactions
- Transaction manager
- Transaction coordinator
- System failure modes
- Network partition
- Commit protocols
- Two-phase commit protocol (2PC)
  - Ready state
  - In-doubt transactions
  - Blocking problem
- Distributed consensus
- Three-phase commit protocol (3PC)
- Persistent messaging
- Concurrency control
- Single lock manager
- Distributed lock manager
- Deadlock handling
  - Local wait-for graph
  - Global wait-for graph
  - False cycles
- Lock leases
- Timestamping
- Replicated data
- Linearizability
- Protocols for replicas
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus protocol
- Robustness
  - Majority-based approach
  - Read one, write all
  - Read one, write all available
  - Node/Site reintegration
- External consistency
- Commit wait
- CAP theorem
- BASE properties
- Asynchronous replication
- Lazy propagation
- Master – slave replication
- Multimaster (update-anywhere) replication
- Asynchronous view maintenance
- Eventual consistency
- Version-vector scheme
- Merkle tree
- Coordinator selection
- Backup coordinator
- Election algorithms
- Bully algorithm
- Term
- Distributed consensus protocol
- Paxos
  - Proposers
  - Acceptors
  - Learners
- Raft

Telegram: @uni_k

www.konkur.in

- Leaders
- Followers
- Replicated state machine
- Fault tolerant lock manager
- Non-blocking two-phase commit

Practice Exercises

23.1 What are the key differences between a local-area network and a wide-area network, that affect the design of a distributed database?

23.2 To build a highly available distributed system, you must know what kinds of failures can occur.

a. List possible types of failure in a distributed system.

b. Which items in your list from part a are also applicable to a centralized system?

23.3 Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Exercise 23.2a, explain how 2PC ensures transaction atomicity despite the failure.

23.4 Consider a distributed system with two sites, A and B. Can site A distinguish among the following?

- B goes down.
- The link between A and B goes down.
- B is extremely overloaded and response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

23.5 The persistent messaging scheme described in this chapter depends on timestamps. A drawback is that they can discard received messages only if they are too old, and may need to keep track of a large number of received messages. Suggest an alternative scheme based on sequence numbers instead of timestamps, that can discard messages more rapidly.

23.6 Explain the difference between data replication in a distributed system and the maintenance of a remote backup site.

23.7 Give an example where lazy replication can lead to an inconsistent database state even when updates get an exclusive lock on the primary (master) copy if data were read from a node other than the master.

23.8 Consider the following deadlock-detection algorithm. When transaction T_i, at site S_1, requests a resource from T_j, at site S , a request message with time-

Telegram: @uni_k

www.konkur.in

stamp n is sent. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_1. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_3 only if T_j has received the request message and cannot immediately grant the requested resource. A request from T_i to T_j in the same site is handled in the usual manner; no timestamps are associated with the edge (T_i, T_j). A central coordinator invokes the detection algorithm by sending an initiating message to each site in the system.

On receiving this message, a site sends its local wait-for graph to the coordinator. Note that such a graph contains all the local information that the site has about the state of the real graph. The wait-for graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.

When the controller has received a reply from each site, it constructs a graph as follows:

- The graph contains a vertex for every transaction in the system.
- The graph has an edge (T_i, T_j) if and only if:
  - There is an edge (T_i, T_j) in one of the wait-for graphs.
  - An edge (T_i, T_j, n) (for some n) appears in more than one wait-for graph.

Show that, if there is a cycle in the constructed graph, then the system is in a deadlock state, and that, if there is no cycle in the constructed graph, then the system was not in a deadlock state when the execution of the algorithm began.

23.9 Consider the chain-replication protocol, described in Section 23.4.3.2, which is a variant of the primary-copy protocol.

a. If locking is used for concurrency control, what is the earliest point when a process can release an exclusive lock after updating a data item?

b. While each data item could have its own chain, give two reasons it would be preferable to have a chain defined at a higher level, such as for each partition or tablet.

c. How can consensus protocols be used to ensure that the chain is uniquely determined at any point in time?

23.10 If the primary copy scheme is used for replication, and the primary gets disconnected from the rest of the system, a new node may get elected as primary. But the old primary may not realize it has got disconnected, and may get reconnected subsequently without realizing that there is a new primary.

a. What problems can arise if the old primary does not realize that a new one has taken over?

b. How can leases be used to avoid these problems?

Telegram: @uni_k

www.konkur.in

c. Would such a situation, where a participant node gets disconnected and then reconnected without realizing it was disconnected, cause any problem with the majority or quorum protocols?

23.11 Consider a federated database system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

a. Suggest ways in which the federated database system can ensure that there is at most one active global transaction at any time.

b. Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

23.12 Consider a federated database system in which every local site ensures local serializability, and all global transactions are read only.

a. Show by example that nonserializable executions may result in such a system.

b. Show how you could use a ticket scheme to ensure global serializability.

23.13 Suppose you have a large relation $r(A, B, C)$ and a materialized view $v = {}_A\gamma_{\text{sum}(B)}(r)$. View maintenance can be performed as part of each transaction that updates r, on a parallel/distributed storage system that supports transactions across multiple nodes. Suppose the system uses two-phase commit along with a consensus protocol such as Paxos, across geographically distributed data centers.

a. Explain why it is not a good idea to perform view maintenance as part of the update transaction, if some values of attribute A are "hot" at certain points in time, that is, many updates pertain to those values of A.

b. Explain how operation locking (if supported) could solve this problem.

c. Explain the tradeoffs of using asynchronous view maintenance in this context.

Exercises

23.14 What characteristics of an application make it easy to scale the application by using a key-value store, and what characteristics rule out deployment on key-value stores?

23.15 Give an example where the read one, write all available approach leads to an erroneous state.

Telegram: @uni_k

www.konkur.in

23.16 In the majority protocol, what should the reader do if it finds different values from different copies, to (a) decide what is the correct value, and (b) to bring the copies back to consistency? If the reader does not bother to bring the copies back to consistency, would it affect correctness of the protocol?

23.17 If we apply a distributed version of the multiple-granularity protocol of Chapter 18 to a distributed database, the site responsible for the root of the DAG may become a bottleneck. Suppose we modify that protocol as follows:

- Only intention-mode locks are allowed on the root.
- All transactions are given the strongest intention-mode lock (IX) on the root automatically.

Show that these modifications alleviate this problem without allowing any non-serializable schedules.

23.18 Discuss the advantages and disadvantages of the two methods that we presented in Section 23.3.4 for generating globally unique timestamps.

23.19 Spanner provides read-only transactions a snapshot view of data, using multi-version two-phase locking.

a. In the centralized multi-version 2PL scheme, read-only transactions never wait. But in Spanner, reads may have to wait. Explain why.

b. Using an older timestamp for the snapshot can reduce waits, but has some drawbacks. Explain why, and what the drawbacks are.

23.20 Merkle trees can be made short and fat (like B^+-trees) or thin and tall (like binary search trees). Which option would be better if you are comparing data across two sites that are geographically separated, and why?

23.21 Why is the notion of term important when an election is used to choose a coordinator? What are the analogies between elections with terms and elections used in a democracy?

23.22 For correct execution of a replicated state machine, the actions must be deterministic. What could happen if an action is non-deterministic?

Further Reading

Textbook coverage of distributed transaction processing, including concurrency control and the two-phase and three-phase commit protocols, is provided by [Bernstein and Goodman (1981)] and [Bernstein and Newcomer (2009)]. Textbook discussions of distributed databases are offered by [Ozsu and Valduriez (2010)]. A collection of papers on data management on cloud systems is in [Ooi and Parthasarathy (2009)].

Telegram: @uni_k

www.konkur.in

The implementation of the transaction concept in a distributed database is presented by [Gray (1981)] and [Traiger et al. (1982)]. The 2PC protocol was developed by [Lampson and Sturgis (1976)]. The three-phase commit protocol is from [Skeen (1981)]. Techniques for non-blocking two-phase commit based on consensus, called Paxos Commit, are described in [Gray and Lamport (2004)].

Chain replication was initially proposed by [van Renesse and Schneider (2004)] and an optimized version of was proposed by [Terrace and Freedman (2009)].

Distributed optimistic concurrency control is described in [Agrawal et al. (1987)], while distributed snapshot isolation is described in [Binnig et al. (2014)] and [Schenkel et al. (1999)]. The externally consistent distributed multi-version 2PL scheme used in Spanner is described in [Corbett et al. (2013)].

The CAP theorem was conjectured by [Brewer (2000)], and was formalized and proved by [Gilbert and Lynch (2002)]. [Cooper et al. (2008)] describe Yahoo!'s PNUTS system, including its support for asynchronous maintenance of replicas using a publish-subscribe system. Parallel view maintenance is described in [Chen et al. (2004)] and [Zhang et al. (2004)], while asynchronous view maintenance is described in [Agrawal et al. (2009)]. Transaction processing in federated database systems is discussed in [Mehrotra et al. (2001)].

Paxos is described in [Lamport (1998)]; Paxos is based on features from several earlier protocols, reference in [Lamport (1998)]. Google's Chubby lock service, which is based on Paxos, is described by [Burrows (2006)]. The widely used ZooKeeper system for distributed coordination is described in [Hunt et al. (2010)], and the consensus protocol (also known as atomic broadcast protocol) used in ZooKeeper is described in [Junqueira et al. (2011)]. The Raft consensus protocol is described in [Ongaro and Ousterhout (2014)].

Bibliography

[Agrawal et al. (1987)] D. Agrawal, A. Bernstein, P. Gupta, and S. Sengupta, "Distributed optimistic concurrency control with reduced rollback", *Distributed Computing*, Volume 2, Number 1 (1987), pages 45–59.

[Agrawal et al. (2009)] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for VLSD databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2009), pages 179–192.

[Bernstein and Goodman (1981)] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Volume 13, Number 2 (1981), pages 185–221.

[Bernstein and Newcomer (2009)] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd edition, Morgan Kaufmann (2009).

[Binnig et al. (2014)] C. Binnig, S. Hildenbrand, F. FÃ¤rber, D. Kossmann, J. Lee, and N. May, "Distributed snapshot isolation: global

Telegram: @uni_k

www.konkur.in

pay locally", *VLDB Journal*, Volume 23, Number 6 (2014), pages 987–1011.

[Brewer (2000)] E. A. Brewer, "Towards Robust Distributed Systems (Abstract)", In *Proc. of the ACM Symposium on Principles of Distributed Computing* (2000), page 7.

[Burrows (2006)] M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems", In *Symp. on Operating Systems Design and Implementation (OSDI)* (2006), pages 335–350.

[Chen et al. (2004)] S. Chen, B. Liu, and E. A. Rundensteiner, "Multiversion-based view maintenance over distributed data sources", *ACM Transactions on Database Systems*, Volume 29, Number 4 (2004), pages 675–709.

[Cooper et al. (2008)] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform", *Proceedings of the VLDB Endowment*, Volume 1, Number 2 (2008), pages 1277–1288.

[Corbett et al. (2013)] J. C. Corbett et al., "Spanner: Google's Globally Distributed Database", *ACM Trans. on Computer Systems*, Volume 31, Number 3 (2013).

[Gilbert and Lynch (2002)] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", *SIGACT News*, Volume 33, Number 2 (2002), pages 51–59.

[Gray (1981)] J. Gray, "The Transaction Concept: Virtues and Limitations", In *Proc. of the International Conf. on Very Large Databases* (1981), pages 144–154.

[Gray and Lamport (2004)] J. Gray and L. Lamport, "Consensus on transaction commit", *ACM Transactions on Database Systems*, Volume 31, Number 1 (2004), pages 133–160.

[Hunt et al. (2010)] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems", In *USENIX Annual Technical Conference (USENIX ATC)* (2010), pages 11–11.

[Junqueira et al. (2011)] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems", In *IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)* (2011), pages 245–256.

[Lamport (1998)] L. Lamport, "The Part-Time Parliament", *ACM Trans. Comput. Syst.*, Volume 16, Number 2 (1998), pages 133–169.

[Lampson and Sturgis (1976)] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System", Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto (1976).

[Mehrotra et al. (2001)] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz, "Overcoming Heterogeneity and Autonomy in Multidatabase Systems.", *Inf. Comput.*, Volume 167, Number 2 (2001), pages 137–172.

[Ongaro and Ousterhout (2014)] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm", In *USENIX Annual Technical Conference (USENIX ATC)* (2014), pages 305–319.

Telegram: @uni_k

www.konkur.in

[Ooi and Parthasarathy (2009)] B. C. Ooi and S. Parthasarathy, "Special Issue on Data Management on Cloud Computing Platforms", *IEEE Data Engineering Bulletin*, Volume 32, Number 1 (2009).

[Ozsu and Valduriez (2010)] T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd edition, Prentice Hall (2010).

[Schenkel et al. (1999)] R. Schenkel, G. Weikum, N. Weisenberg, and X. Wu, "Federated Transaction Management with Snapshot Isolation", In *Eight International Workshop on Foundations of Models and Languages for Data and Objects, Transactions and Database Dynamics* (1999), pages 1–25.

[Skeen (1981)] D. Skeen, "Non-blocking Commit Protocols", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), pages 133–142.

[Terrace and Freedman (2009)] J. Terrace and M. J. Freedman, "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads", In *USENIX Annual Technical Conference (USENIX ATC)* (2009).

[Traiger et al. (1982)] I. L. Traiger, J. N. Gray, C. A. Galtieri, and B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems", *ACM Transactions on Database Systems*, Volume 7, Number 3 (1982), pages 323–342.

[van Renesse and Schneider (2004)] R. van Renesse and F. B. Schneider, "Chain Replication for Supporting High Throughput and Availability", In *Symp. on Operating Systems Design and Implementation (OSDI)* (2004), pages 91–104.

[Zhang et al. (2004)] X. Zhang, L. Ding, and E. A. Rundensteiner, "Parallel multisource view maintenance", *VLDB Journal*, Volume 13, Number 1 (2004), pages 22–48.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in



PART 9

ADVANCED TOPICS

Chapter 24 provides further details about the index structures we covered in Chapter 14. In particular, this chapter provides detailed coverage of the LSM tree and its variants, bitmap indices, and spatial indexing, all of which were covered in brief in Chapter 14. The chapter also provides detailed coverage of dynamic hashing techniques.

Chapter 25 discusses a number of tasks involved in application development. Applications can be made to run significantly faster by performance tuning, which consists of finding and eliminating bottlenecks and adding appropriate hardware such as memory or disks. Application performance is evaluated using benchmarks, which are standardized sets of tasks that help to characterize the performance of database systems. Another important aspect of application development is testing, which requires the generation of database states and test inputs, followed by checking that the actual outputs of a query or a program on the test input match the expected outputs. Lastly, standards are very important for application development. A variety of standards have been proposed that affect database-application development. We outline several of these standards in this chapter.

Chapter 26 covers blockchain technology from a database perspective. This chapter identifies the ways in which blockchain databases differ from the traditional databases covered elsewhere in this text and shows how these distinguishing features are implemented. Although blockchain systems are often associated with Bitcoin, this chapter goes beyond Bitcoin-style algorithms and implementation to focus on alternatives that are more suited to an enterprise database environment.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 24



Advanced Indexing Techniques

We studied the concept of indexing, as well as a number of different index structures in Chapter 14. While some index structures, such as B^+-trees, were covered in detail, others such as hashing, write-optimized indices, bitmap indices, and spatial indices were only briefly outlined in Chapter 14. In this chapter we provide further details of these index structures. We provide detailed coverage of the LSM tree and its variants. We then provide a detailed description of bitmap indices. Next, we provide more detailed coverage of spatial indexing, covering quad trees and R-trees in more detail. Finally, we cover hashing, with detailed coverage of dynamic hashing techniques.

24.1 Bloom Filter

A **Bloom filter** is a probabilistic data structure that can check for membership of a value in a set using very little space, but at a small risk of overestimating the set of elements that are in the set. A Bloom filter is basically a bitmap. If the set has n values, the associated bitmap has a few times n (typically $10n$) bits; the Bloom filter also has associated with it several hash functions. We assume initially that there is only one hash function $h()$.

The bits in the bitmap are all initially set to 0; subsequently, each value in the set is read, and the hash function $h(v)$ is computed on the element v, with the range of the function being 1 to $10n$. The bit at position $h(v)$ is then set to 1. This is repeated for every element v. To check if a particular value v is present in the set, the hash function $h(v)$ is computed. If bit $h(v)$ in the Bloom filter is equal to 0, we can infer that v cannot possibly be in the set. However, if bit $h(v)$ is equal to 1, v may be present in the set. Note that with some probability, the bit $h(v)$ may be 1 even if v is not present, if some other value v', present in the set has $h(v') = h(v)$. Thus, a lookup for v results in a *false positive*.

To reduce the chance of false positives, Bloom filters use k independent hash functions $h_i(), i = 1..k$, for some $k > 1$; for each value v in the set, bits corresponding to $h_i(v), i = 1..k$ are all set to 1 in the bitmap. When querying the Bloom filter with a

Telegram: @uni_k

www.konkur.in

given value v the same k hash functions are used to identify k bit locations; the value v is determined to be absent if even one of these bits has a 0 value. Otherwise the value is judged to be potentially present. For example, if the bitmap has $10n$ bits, where n is the number of values in the set, and $k = 7$ hash functions are used, the false positive rate would be about 1%.

24.2 Log-Structured Merge Tree and Variants

As we saw in Section 14.8, B^+-tree indices are not efficient for workloads with a very high number of writes, and alternative index structures have been proposed to handle such workloads. We saw a brief description of two such index structures, the *log-structured merge tree* or LSM tree and its variants, in Section 14.8.1, and the *buffer tree*, in Section 14.8.2. In this section we provide further details of the LSM tree and its variants. To help with the discussions, we repeat some of the basic material we presented in Section 14.8.1.

The key idea of the **log-structured merge tree** (**LSM tree**) is to replace random I/O operations during tree inserts, updates, and deletes with a smaller number of sequential I/O operations. Our initial description focuses on index inserts and lookups; we describe how to handle updates and deletes later in the section.

An LSM tree consists of several B^+-trees, starting with an in-memory tree, called L_0, and on-disk trees $L_1, L_2, \ldots, L_k$ for some k, where k is called the level. Figure 24.1 depicts the structure of an LSM tree for $k = 3$.

An index lookup is performed by using separate lookup operations on each of the trees $L_0, \ldots, L_k$, and merging the results of the lookups. (We assume here that there are no updates or deletes; we will discuss how to perform lookups in the presence of updates/deletes later.)

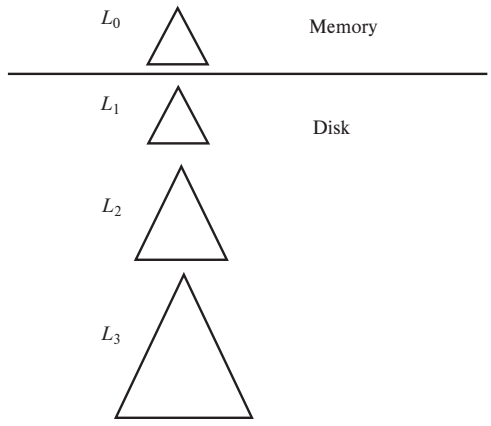


Figure 24.1 Log-structured merge tree with three levels.

Telegram: @uni_k

www.konkur.in

24.2.1 Insertion into LSM Trees

When a record is first inserted into an LSM tree, it is inserted into the in-memory B$^+$-tree structure L_0. A fairly large amount of memory space is allocated for this tree. As the tree grows to fill the memory allocated to it, we need to move data from the in-memory structure to a B$^+$-tree on disk.

If tree L_1 is empty, the entire in-memory tree L_0 is written to disk to create the initial tree L_1. However, if L_1 is not empty, the leaf level of L_0 is scanned in increasing key order, and entries are merged with the leaf level entries of L_1 (also scanned in increasing key order). The merged entries are used to create a new B$^+$-tree using the bottom-up build process. The new tree with the merged entries then replaces the old L_1. In either case, after entries of L_0 have been moved to L_1, all entries in L_0 are deleted. Inserts can then be made to the now empty L_0.

Note that all entries in the leaf level of the old L_1 tree, including those in leaf nodes that do not have any updates, are copied to the new tree instead of being inserted into the existing L_1 tree node. This gives the following benefits:

- The leaves of the new tree are sequentially located, avoiding random I/O during subsequent merges.
- The leaves are full, avoiding the overhead of partially occupied leaves that can occur with page splits.

There is, however, a cost to using the LSM structure: the entire contents of the tree are copied each time a set of entries from L_0 are copied into L_1.

If the tree structure is implemented on top of a distributed file system (Section 21.6), copying data to a new tree is often unavoidable, since most distributed file systems do not support updates to an already created block.

To ensure we get a benefit for cases where the index size on disk is much bigger than the in-memory index, the maximum size of L_1 is chosen as k times the target size of L_0, for some k. Similarly, the maximum size of each L_{i+1} is set to k times the target size of L_i. Once a particular L_i reaches its maximum size, its entries are merged into the next component L_{i+1}. When L_{i+1} reaches its target size, its entries are in turn merged into L_{i+2}, and so on.

Note that if each leaf of L_i has m entries, m/k entries would map to a single leaf node of L_{i+1}. The value of k is chosen to ensure that m/k is some reasonable number, say 10. Let M denote the size of L_0. Then, the size of a tree at level L_i is k^iM. The total number of levels r is thus roughly $log_k(I/M)$ where I is the total size of the index entries.

Let us now consider the number of I/O operations required with a multiple-level LSM tree. At each L_i, m/k inserts are performed using only one I/O operation. On the other hand, each entry gets inserted once at each level L_i. Thus, the total number of I/O operations for each insert is $(k/m)log_k(I/M)$. Thus, as long as the number of

Telegram: @uni_k

www.konkur.in

levels $r = log_k(I/M)$ is less than m/k, the overall number of I/O operations per insert is reduced by using an LSM tree as compared to direct insertion into a B^+-tree.

If, for example, $r = 10$ and $k = 10$, the in-memory index needs to be greater than 1% of the total index size to get a benefit in terms of the number of I/O operations required for inserts. (As before, the benefit of reduced seeks is available even if the L_0 is significantly smaller.)

If the number of levels is greater than m/k, even if there is no benefit in terms of number of I/O operations, there can still be savings since sequential I/O is used instead of random I/O. In Section 24.2.4 we describe a variant of the LSM tree which further reduces the overhead on write operations, at the cost of adding overhead on read operations.

One way to avoid creating large LSM trees with many levels is to *range partition* the relation and create separate LSM trees on each partition. Such partitioning is natural in a parallel environment, as we saw earlier in Section 21.2. In particular, in such environments, a partition can be dynamically repartitioned into smaller pieces whenever it becomes too large, as we saw in Section 21.3.3. With such repartitioning, the size of each LSM tree can be kept small enough to avoid having a large number of levels. There is a price for such partitioning: each partition requires its own L_0 tree in memory. As a result, although it can be used in a centralized setting, the partitioning approach best fits a parallel environment where resources such as processing nodes can be added as the load increases.

24.2.2 Rolling Merges

We assumed for simplicity that when a particular level is full, its entries are entirely merged with the next level. This would result in more I/O load during merges with an unused I/O capacity between merges. To avoid this problem, merging is done on a continuous basis; this is called rolling merge.

With **rolling merge**, a few pages of L_i are merged into corresponding pages of L_{i+1} at a time, and removed from L_i. This is done whenever L_i becomes close to its target size, and it results in L_i shrinking a bit to return to its target size. When L_i grows again, the rolling merge restarts from a point at the leaf level of L_i just after where the earlier rolling merge stopped, so the scan is sequential. When the end of the L_i tree is reached, the scan starts again at the beginning of the tree. Such a merge is called a rolling merge since records are moved from one level to another on a continuous basis.

The number of leaves merged at a time is kept high enough to ensure that the seek time is small compared to the time to transfer data from and to disk.

24.2.3 Handling Deletes and Updates

So far we have only described inserts and lookups. Deletes are handled in an interesting manner. Instead of directly finding an index entry and deleting it, deletion results in insertion of a new **deletion entry** that indicates which index entry is to be deleted. The

Telegram: @uni_k

www.konkur.in

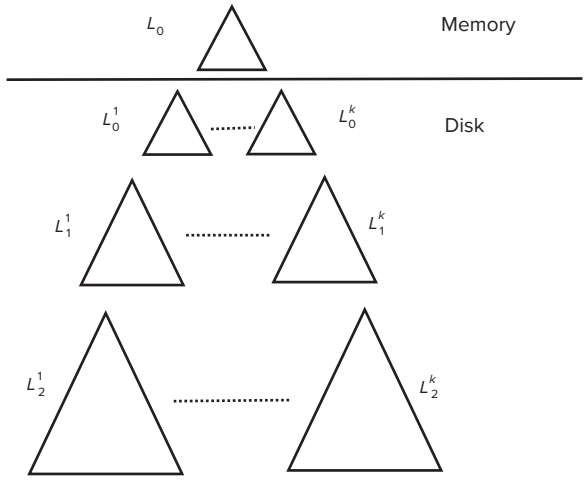


Figure 24.2 stepped-merge index

process of inserting a deletion entry is identical to the process of inserting a normal index entry.

However, lookups have to carry out an extra step. As mentioned earlier, lookups retrieve entries from all the trees and merge them in sorted order of key value. If there is a deletion entry for some entry, both of them would have the same key value. Thus, a lookup would find both the deletion entry and the original entry for that key, which is to be deleted. If a deletion entry is found, the to-be-deleted entry should be filtered out and not returned as part of the lookup result.

When trees are merged, if one of the trees contains an entry, and the other had a matching deletion entry, the entries get matched up during the merge (both would have the same key), and are both discarded.

Updates are handled in a manner similar to deletes, by inserting an update entry. Lookups need to match update entries with the original entries and return the latest value. The update is actually applied during a merge, when one tree has an entry and another has its matching update entry; the update is applied during the merge, and the update entry is discarded.

24.2.4 The Stepped-Merge Index

We now consider a variant of the LSM tree, which has multiple trees at each level instead of one tree per level and performs inserts in a slightly different manner. This structure is shown in Figure 24.2. We call the structure a **stepped-merge index**, following the terminology in an early paper that introduced it. In the developer community, the basic LSM tree, the stepped-merge index, and several other variants are all referred to as LSM trees. We use the terms *stepped-merge index* and *basic LSM tree* to clearly identify which variant we are referring to.

Telegram: @uni_k

www.konkur.in

24.2.4.1 Insertion Algorithm

In the stepped-merge index, incoming data are initially stored in memory, in an L_0 tree, in a manner similar to the LSM tree. However, when the tree reaches its maximum size, instead of merging it into an L_1 tree, the in-memory L_0 tree is written to disk. When the in-memory tree again reaches its maximum size, it is again written to disk. Thus, we may have multiple L_0 trees on disk, which we shall refer to as L_0^1, L_0^2 and so forth. Each of the L_0^i trees is a B$^+$-tree and can be written to disk using only sequential I/O operations.

If this process is repeated, after a while we would have a large number of trees, each as large as memory, stored on disk. Lookups would then have to pay a high price, since they would have to search through each of the tree structures, incurring separate I/O costs on each search.

To limit the overhead on lookups, once the number of on-disk trees at a level L_i reaches some limit k, all the trees at a level are merged together into one combined new tree structure at the next level L_{i+1}. The leaves of the trees at level L_i are read sequentially, and the keys merged in sorted order, and the level L_{i+1} tree is constructed using standard techniques for bottom-up construction of B$^+$-trees. As before, the merge operation avoids random I/O operations, since it reads the individual tree structures sequentially and writes the resultant merged tree also sequentially.

Once a set of trees are merged into a single new tree, future queries can search the merged tree; the original trees can then be deleted (after ensuring any ongoing searches have completed).

The benefit of the stepped-merge index scheme as compared to the basic LSM tree is that index entries are written out only once per level. With the basic LSM tree, each time a tree at level L_i is merged into a tree at level L_{i+1}, the entire contents of the L_{i+1} tree is read and written back to disk. Thus, on average, each record is read and written back $k/2$ times at each level of an LSM tree for a total of $klog_k(I/M)$ I/O operations. In contrast, with stepped-merge index, each record is written to disk once per layer, and read again when merging into the next layer, for a total of approximately $2log_k(I/M)$ I/O operations. Thus, the stepped-merge index incurs significantly less cost for updates.

The total number of bytes written (across all levels) on account of inserting an entry, divided by the size of entry, is called the **write amplification**. To calculate the write amplification of the LSM trees and the stepped-merge index, we can modify the above formulae for I/O operations by ignoring the reads. For a B$^+$-tree where each leaf gets on average only one update before it is written back, the write amplification would be the size of the page divided by the size of the index entry.

For a B$^+$-tree, if a page has 100 entries, the write amplification would be 100. With $k = 5$, and $I = 100M$, we would have $log_5(100) = 3$ levels. The write amplification of an LSM tree would then be $5/2 \times 3 = 7.5$. The write amplification of the stepped-merge index would be 3. With $k = 10$, the tree would have $log_{10}(100) = 2$ levels, leading to a write amplification of 2 for stepped-merge index, and 10 for an LSM tree.

Telegram: @uni_k

www.konkur.in

Note that like the basic LSM tree, the stepped-merge index also requires no random I/O operations during insertion, in contrast to a B^+-tree insertion. Thus, the performance of B^+-trees would be worse than what the write amplification number above indicates.

Merging can be optimized as follows: While merging k trees at a particular level L_i, into a level L_{i+1} tree, trees at levels $L_j, j < i$ can also be merged in at the same time. Entries in these trees can thus entirely skip one or more levels of the stepped-merge index. Further, if the system has idle capacity, trees at a level L_i can be merged even if there are fewer than k trees at that level. In a situation where there is a long period of time with very few inserts, and the system load is light, trees across all levels could potentially get merged into a single tree at some level r.

24.2.4.2 Lookup Operations Using Bloom Filters

Lookup operations in stepped-merge index have to separately search each of the trees. Thus, compared to the basic LSM scheme, the stepped-merge index increases the burden on lookups, since in the worst case lookups need to access k trees at each level, leading to a total of $k * log_k(I/M)$ tree lookups, instead of $log_k(I/M)$ tree lookups in the worst case with a basic LSM tree.

For workloads with a significant fraction of reads, this overhead can be unacceptable. For example, with the stepped-merge index with $I = 100M$ and $k = 5$, a single lookup requires 15 tree traversals, while the LSM tree would require 3 tree traversals. Note that for the common case where each key value occurs in only one tree, only one of the traversals would find a given search key, while all the other traversals would fail to find the key.

To reduce the cost of point lookups (i.e., lookups of a given key value), most systems use a Bloom filter to check if a tree can possibly contain the given key value. One Bloom filter is associated with each tree, and it is built on the set of key values in the tree. To check if a particular tree may contain a search key v, the key v is looked up in the Bloom filter. If the Bloom filter indicates that the key value is absent, it is definitely not present in the tree, and lookup can skip that tree. Otherwise, the key value may be present in the tree, which must be looked up.

A Bloom filter with $10n$ bits, where the tree has n elements, and using 7 hash functions would give a false positive rate of 1 percent. Thus, for a lookup on a key that is present in the index, on average just slightly more than one tree would be accessed. Thus, lookup performance would be only slightly worse than on a regular B^+-tree.

The Bloom filter check thus works very well for point lookups, allowing a significant fraction of the trees to be skipped, as long as sufficient memory is available to store all the Bloom filters in memory. With I key values in the index, approximately $10I$ bits of memory will be required. To reduce the main memory overhead, some of the Bloom filters may be stored on flash storage.

Note that for range lookups, the Bloom filter optimization cannot be used, since there is no unique hash value. Instead, all trees must be accessed separately.

Telegram: @uni_k

www.konkur.in

24.2.5 LSM Trees For Flash Storage

LSM trees were initially designed to reduce the write and seek overheads of hard disks. Flash disks have a relatively low overhead for random I/O operations since they do not require seek, and thus the benefit of avoiding random I/O that LSM tree variants provide is not particularly important with flash disks.

However, recall that flash memory does not allow in-place update, and writing even a single byte to a page requires the whole page to be rewritten to a new physical location; the original location of the page needs to be erased eventually, which is a relatively expensive operation. The reduction in write amplification using LSM tree variants, as compared to traditional B^+-trees, can provide substantial performance benefits when LSM trees are used with flash storage.

24.3 Bitmap Indices

As we saw in Section 14.9, a bitmap index is a specialized type of index designed for easy querying on multiple keys. Bitmaps work best for attributes that take only a small number of distinct values.

For bitmap indices to be used, records in a relation must be numbered sequentially, starting from, say, 0. Given a number n, it must be easy to retrieve the record numbered n. This is particularly easy to achieve if records are fixed in size and allocated on consecutive blocks of a file. The record number can then be translated easily into a block number and a number that identifies the record within the block.

Recall that *column-oriented storage*, described in Section 13.6, stores attributes in arrays, allowing efficient access of the attribute of the ith record, for any given i. Bitmap indices are thus particularly useful with columnar storage.

We use as a running example a relation *instructor_info*, which has an attribute *gender*, which can take only values m (male) or f (female), and an attribute *income_level*, where income has been broken up into 5 levels: $L1$: 0-9999, $L2$: 10,000-19,999, $L3$: 20,000-39,999, $L4$: 40,000-74,999, and $L5$: $75,000 - \infty$.

24.3.1 Bitmap Index Structure

As we saw in Section 14.9, a **bitmap** is simply an array of bits. In its simplest form, a **bitmap index** on the attribute A of relation r consists of one bitmap for each value that A can take. Each bitmap has as many bits as the number of records in the relation. The ith bit of the bitmap for value v_j is set to 1 if the record numbered i has the value v_j for attribute A. All other bits of the bitmap are set to 0.

In our example, there is one bitmap for the value m and one for f. The ith bit of the bitmap for m is set to 1 if the *gender* value of the record numbered i is m. All other bits of the bitmap for m are set to 0. Similarly, the bitmap for f has the value 1 for bits corresponding to records with the value f for the *gender* attribute; all other bits have

Telegram: @uni_k

www.konkur.in

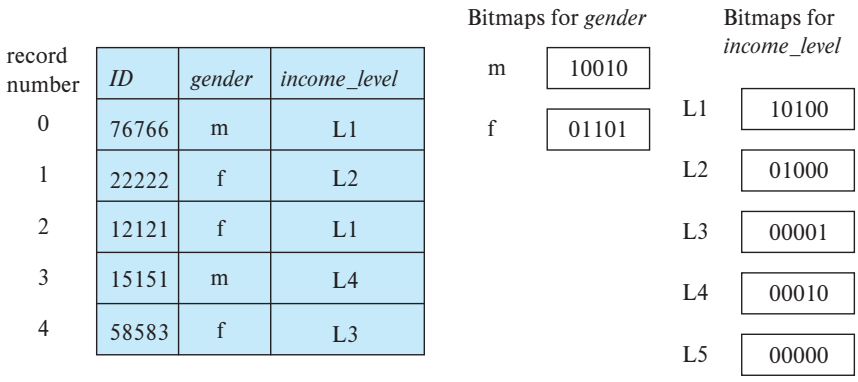


Figure 24.3 Bitmap indices on relation *instructor_info*.

the value 0. Figure 24.3 shows an example of bitmap indices on a relation *instructor_info*

We now consider when bitmaps are useful. The simplest way of retrieving all records with value m (or value f) would be to simply read all records of the relation and select those records with value m (or f, respectively). The bitmap index doesn't really help to speed up such a selection. While it would allow us to read only those records for a specific gender, it is likely that every disk block for the file would have to be read anyway.

In fact, bitmap indices are useful for selections mainly when there are selections on multiple keys. Suppose we create a bitmap index on attribute *income_level*, which we described earlier, in addition to the bitmap index on *gender*.

Consider now a query that selects women with income in the range \$10,000 *to* \$19, 999. This query can be expressed as

```
select *
from instructor_info
where gender = 'f' and income_level = 'L2';
```

To evaluate this selection, we fetch the bitmaps for *gender* value f and the bitmap for *income_level* value *L2* and perform an **intersection** (logical-and) of the two bitmaps. In other words, we compute a new bitmap where bit *i* has value 1 if the *i*th bit of the two bitmaps are both 1 and has a value 0 otherwise. In the example in Figure 24.3, the intersection of the bitmap for *gender* = f (01101) and the bitmap for *income_level* = *L2* (01000) gives the bitmap 01000.

Since the first attribute can take two values, and the second can take five values, we would expect only about 1 in 10 records, on an average, to satisfy a combined condition on the two attributes. If there are further conditions, the fraction of records satisfying all the conditions is likely to be quite small. The system can then compute the query result by finding all bits with value 1 in the intersection bitmap and retrieving the cor-

Telegram: @uni_k

www.konkur.in

responding records. If the fraction is large, scanning the entire relation would remain the cheaper alternative.

Another important use of bitmaps is to count the number of tuples satisfying a given selection. Such queries are important for data analysis. For instance, if we wish to find out how many women have an income level $L2$, we compute the intersection of the two bitmaps and then count the number of bits that are 1 in the intersection bitmap. We can thus get the desired result from the bitmap index, without even accessing the relation.

Bitmap indices are generally quite small compared to the actual relation size. Records are typically at least tens of bytes to hundreds of bytes long, whereas a single bit represents the record in a bitmap. Thus, the space occupied by a single bitmap is usually less than 1 percent of the space occupied by the relation. For instance, if the record size for a given relation is 100 bytes, then the space occupied by a single bitmap will be $\frac{1}{8}$ of 1 percent of the space occupied by the relation. If an attribute A of the relation can take on only one of eight values, a bitmap index on attribute A would consist of eight bitmaps, which together occupy only 1 percent of the size of the relation.

Deletion of records creates gaps in the sequence of records, since shifting records (or record numbers) to fill gaps would be extremely expensive. To recognize deleted records, we can store an **existence bitmap**, in which bit i is 0 if record i does not exist and 1 otherwise. We shall see the need for existence bitmaps in Section 24.3.2. Insertion of records should not affect the sequence numbering of other records. Therefore, we can do insertion either by appending records to the end of the file or by replacing deleted records.

24.3.2 Efficient Implementation of Bitmap Operations

We can compute the intersection of two bitmaps easily by using a **for** loop: the ith iteration of the loop computes the **and** of the ith bits of the two bitmaps. We can speed up computation of the intersection greatly by using bit-wise **and** instructions supported by most computer instruction sets. A *word* usually consists of 32 or 64 bits, depending on the architecture of the computer. A bit-wise **and** instruction takes two words as input and outputs a word where each bit is the logical **and** of the bits in corresponding positions of the input words. What is important to note is that a single bit-wise **and** instruction can compute the intersection of 32 or 64 bits *at once*.

If a relation had 1 million records, each bitmap would contain 1 million bits, or equivalently 128 kilobytes. Only 31,250 instructions are needed to compute the intersection of two bitmaps for our relation, assuming a 32-bit word length. Thus, computing bitmap intersections is an extremely fast operation.

Just as bitmap intersection is useful for computing the **and** of two conditions, bitmap union is useful for computing the **or** of two conditions. The procedure for bitmap union is exactly the same as for intersection, except we use bit-wise **or** instructions instead of bit-wise **and** instructions.

Telegram: @uni_k

www.konkur.in

The complement operation can be used to compute a predicate involving the negation of a condition, such as **not** (*income-level* = $L1$). The complement of a bitmap is generated by complementing every bit of the bitmap (the complement of 1 is 0 and the complement of 0 is 1). It may appear that **not** (*income_level* = $L1$) can be implemented by just computing the complement of the bitmap for income level $L1$. If some records have been deleted, however, just computing the complement of a bitmap is not sufficient. Bits corresponding to such records would be 0 in the original bitmap but would become 1 in the complement, although the records don't exist. A similar problem also arises when the value of an attribute is *null*. For instance, if the value of *income_level* is null, the bit would be 0 in the original bitmap for value $L1$ and 1 in the complement bitmap.

To make sure that the bits corresponding to deleted records are set to 0 in the result, the complement bitmap must be intersected with the existence bitmap to turn off the bits for deleted records. Similarly, to handle null values, the complement bitmap must also be intersected with the complement of the bitmap for the value *null*.[1]

Counting the number of bits that are 1 in a bitmap can be done quickly by a clever technique. We can maintain an array with 256 entries, where the ith entry stores the number of bits that are 1 in the binary representation of i. Set the total count initially to 0. We take each byte of the bitmap, use it to index into this array, and add the stored count to the total count. The number of addition operations is $\frac{1}{8}$ of the number of tuples, and thus the counting process is very efficient. A large array (using $2^{16} = 65{,}536$ entries), indexed by pairs of bytes, would give even higher speedup, but at a higher storage cost.

24.3.3 Bitmaps and B$^+$-Trees

Bitmaps can be combined with regular B$^+$-tree indices for relations where a few attribute values are extremely common, and other values also occur, but much less frequently. In a B$^+$-tree index leaf, for each value we would normally maintain a list of all records with that value for the indexed attribute. Each element of the list would be a record identifier, consisting of at least 32 bits, and usually more. For a value that occurs in many records, we store a bitmap instead of a list of records.

Suppose a particular value v_i occurs in $\frac{1}{16}$ of the records of a relation. Let N be the number of records in the relation, and assume that a record has a 64-bit number identifying it. The bitmap needs only 1 bit per record, or N bits in total. In contrast, the list representation requires 64 bits per record where the value occurs, or $64 * N/16 = 4N$ bits. Thus, a bitmap is preferable for representing the list of records for value v_i. In our example (with a 64-bit record identifier), if fewer than 1 in 64 records have a particular value, the list representation is preferable for identifying records with that

[1] Handling predicates such as **is unknown** would cause further complications, which would in general require use of an extra bitmap to track which operation results are unknown.

Telegram: @uni_k

www.konkur.in

value, since it uses fewer bits than the bitmap representation. If more than 1 in 64 records have that value, the bitmap representation is preferable.

Thus, bitmaps can be used as a compressed storage mechanism at the leaf nodes of B^+-trees for those values that occur very frequently.

24.4 Indexing of Spatial Data

As we saw in Section 14.10.1, indices are required for efficient access to spatial data, and such indices must efficiently support queries such as range and nearest neighbor queries. We also gave a brief overview of k-d trees, quadtrees, and R-trees; we also briefly described how to answer range queries using k-d trees. In this section we provide further details of quadtrees and R-trees.

As mentioned in Section 14.10.1, in addition to indexing of points, spatial indices must also support indexing of regions of space such as line segments, rectangles, and other polygons. There are extensions of k-d trees and quadtrees for this task. However, a line segment or polygon may cross a partitioning line. If it does, it has to be split and represented in each of the subtrees in which its pieces occur. Multiple occurrences of a line segment or polygon caused by such splits can result in inefficiencies in storage, as well as inefficiencies in querying. R-trees were developed to support efficient indexing of such structures.

24.4.1 Quadtrees

An alternative representation for two-dimensional data are a **quadtree**. An example of the division of space by a quadtree appears in Figure 24.4. Each node of a quadtree is

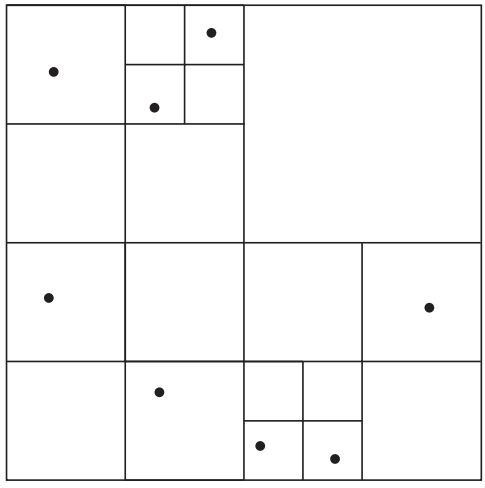


Figure 24.4 Division of space by a quadtree.

Telegram: @uni_k

www.konkur.in

associated with a rectangular region of space. The top node is associated with the entire target space. Each nonleaf node in a quadtree divides its region into four equal-sized quadrants, and correspondingly each such node has four child nodes corresponding to the four quadrants. Leaf nodes have between zero and some fixed maximum number of points. Correspondingly, if the region corresponding to a node has more than the maximum number of points, child nodes are created for that node. In the example in Figure 24.4, the maximum number of points in a leaf node is set to 1.

This type of quadtree is called a **PR quadtree**, to indicate it stores points, and that the division of space is divided based on regions, rather than on the actual set of points stored. We can use **region quadtrees** to store array (raster) information. A node in a region quadtree is a leaf node if all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area and is therefore an internal node. Each node in the region quadtree corresponds to a subarray of values. The subarrays corresponding to leaves either contain just a single array element or have multiple array elements, all of which have the same value.

24.4.2 R-Trees

A storage structure called an **R-tree** is useful for indexing of objects such as points, line segments, rectangles, and other polygons. An R-tree is a balanced tree structure with the indexed objects stored in leaf nodes, much like a B^+-tree. However, instead of a range of values, a rectangular **bounding box** is associated with each tree node. The bounding box of a leaf node is the smallest rectangle parallel to the axes that contains all objects stored in the leaf node. The bounding box of internal nodes is, similarly, the smallest rectangle parallel to the axes that contains the bounding boxes of its child nodes. The bounding box of an object (such as a polygon) is defined, similarly, as the smallest rectangle parallel to the axes that contains the object.

Each internal node stores the bounding boxes of the child nodes along with the pointers to the child nodes. Each leaf node stores the indexed objects and may optionally store the bounding boxes of the objects; the bounding boxes help speed up checks for overlaps of the rectangle with the indexed objects—if a query rectangle does not overlap with the bounding box of an object, it cannot overlap with the object, either. (If the indexed objects are rectangles, there is no need to store bounding boxes, since they are identical to the rectangles.)

Figure 24.5 shows an example of a set of rectangles (drawn with a solid line) and the bounding boxes (drawn with a dashed line) of the nodes of an R-tree for the set of rectangles. Note that the bounding boxes are shown with extra space inside them, to make them stand out pictorially. In reality, the boxes would be smaller and fit tightly on the objects that they contain; that is, each side of a bounding box B would touch at least one of the objects or bounding boxes that are contained in B.

The R-tree itself is at the right side of Figure 24.5. The figure refers to the coordinates of bounding box i as BB_i in the figure.

Telegram: @uni_k

www.konkur.in

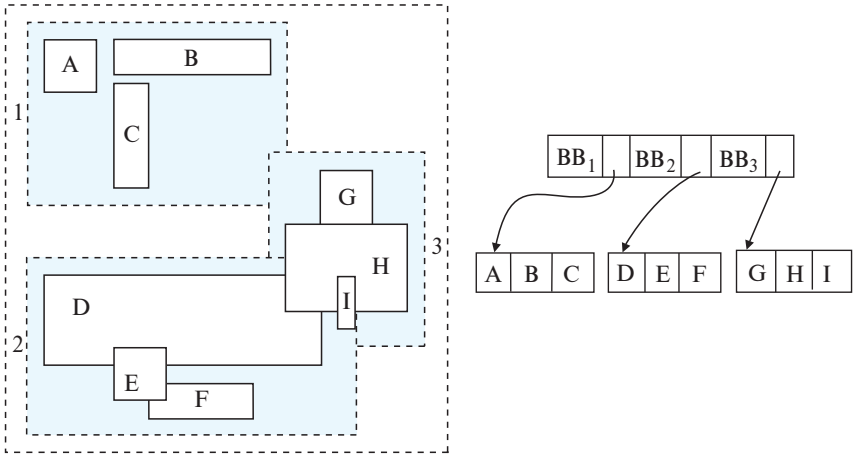


Figure 24.5 An R-tree.

We shall now see how to implement search, insert, and delete operations on an R-tree.

- **Search**. As the figure shows, the bounding boxes associated with sibling nodes may overlap; in B^+-trees, k-d trees, and quadtrees, in contrast, the ranges do not overlap. A search for objects containing a point therefore has to follow *all* child nodes whose associated bounding boxes contain the point; as a result, multiple paths may have to be searched. Similarly, a query to find all objects that intersect a given object has to go down every node where the associated rectangle intersects the given object.

- **Insert**. When we insert an object into an R-tree, we select a leaf node to hold the object. Ideally we should pick a leaf node that has space to hold a new entry, and whose bounding box contains the bounding box of the object. However, such a node may not exist; even if it did, finding the node may be very expensive, since it is not possible to find it by a single traversal down from the root. At each internal node we may find multiple children whose bounding boxes contain the bounding box of the object, and each of these children needs to be explored. Therefore, as a heuristic, in a traversal from the root, if any of the child nodes has a bounding box containing the bounding box of the object, the R-tree algorithm chooses one of them arbitrarily. If none of the children satisfy this condition, the algorithm chooses a child node whose bounding box has the maximum overlap with the bounding box of the object for continuing the traversal.

  Once the leaf node has been reached, if the node is already full, the algorithm performs node splitting (and propagates splitting upward if required) in a manner very similar to B^+-tree insertion. Just as with B^+-tree insertion, the R-tree insertion algorithm ensures that the tree remains balanced. Additionally, it ensures that the

Telegram: @uni_k

www.konkur.in

bounding boxes of leaf nodes, as well as internal nodes, remain consistent; that is, bounding boxes of leaves contain all the bounding boxes of the objects stored at the leaf, while the bounding boxes for internal nodes contain all the bounding boxes of the children nodes.

The main difference of the insertion procedure from the B^+-tree insertion procedure lies in how the node is split. In a B^+-tree, it is possible to find a value such that half the entries are less than the midpoint and half are greater than the value. This property does not generalize beyond one dimension; that is, for more than one dimension, it is not always possible to split the entries into two sets so that their bounding boxes do not overlap. Instead, as a heuristic, the set of entries S can be split into two disjoint sets S_1 and S_2 so that the bounding boxes of S_1 and S_2 have the minimum total area; another heuristic would be to split the entries into two sets S_1 and S_2 in such a way that S_1 and S_2 have minimum overlap. The two nodes resulting from the split would contain the entries in S_1 and S_2, respectively. The cost of finding splits with minimum total area or overlap can itself be large, so cheaper heuristics, such as the *quadratic split* heuristic, are used. (The heuristic gets its name from the fact that it takes time quadratic in the number of entries.)

The **quadratic split** heuristic works this way: First, it picks a pair of entries a and b from S such that putting them in the same node would result in a bounding box with the maximum wasted space; that is, the area of the minimum bounding box of a and b minus the sum of the areas of a and b is the largest. The heuristic places the entries a and b in sets S_1 and S_2, respectively.

It then iteratively adds the remaining entries, one entry per iteration, to one of the two sets S_1 or S_2. At each iteration, for each remaining entry e, let $i_{e,1}$ denote the increase in the size of the bounding box of S_1 if e is added to S_1 and let $i_{e,2}$ denote the corresponding increase for S_2. In each iteration, the heuristic chooses one of the entries with the maximum difference of $i_{e,1}$ and $i_{e,2}$ and adds it to S_1 if $i_{e,1}$ is less than $i_{e,2}$, and to S_2 otherwise. That is, an entry with "maximum preference" for S_1 or S_2 is chosen at each iteration. The iteration stops when all entries have been assigned, or when one of the sets S_1 or S_2 has enough entries that all remaining entries have to be added to the other set so the nodes constructed from S_1 and S_2 both have the required minimum occupancy. The heuristic then adds all unassigned entries to the set with fewer entries.

- **Deletion**. Deletion can be performed like a B^+-tree deletion, borrowing entries from sibling nodes, or merging sibling nodes if a node becomes underfull. An alternative approach redistributes all the entries of underfull nodes to sibling nodes, with the aim of improving the clustering of entries in the R-tree.

See the bibliographical references for more details on insertion and deletion operations on R-trees, as well as on variants of R-trees, called R^*-trees or R^+-trees.

The storage efficiency of R-trees is better than that of k-d trees or quadtrees, since an object is stored only once, and we can ensure easily that each node is at least half

Telegram: @uni_k

www.konkur.in

full. However, querying may be slower, since multiple paths have to be searched. Spatial joins are simpler with quadtrees than with R-trees, since all quadtrees on a region are partitioned in the same manner. However, because of their better storage efficiency and their similarity to B-trees, R-trees and their variants have proved popular in database systems that support spatial data.

24.5 Hash Indices

We described the concepts of hashing and hash indices in Section 14.5. We provide further details in this section.

24.5.1 Static Hashing

As in Section 14.5, let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A hash function h is a function from K to B. Let h denote a hash function. Recall that in a hash index, buckets contain index entries, with pointers to records, while in a hash file organization, actual records are stored in the buckets. All the other details remain the same, so we do not explicitly differentiate between these two versions henceforth. We use the term *hash index* to denote hash file organizations as well as secondary hash indices.

Figure 24.6 shows a secondary hash index on the *instructor* file, for the search key *ID*. The hash function in the figure computes the sum of the digits of the *ID* modulo 8. The hash index has eight buckets, each of size 2 (realistic indices would have much larger bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *ID* is a primary key for *instructor*, so each search key has only one associated pointer. In general, multiple pointers can be associated with each key.

Hash indices can efficiently answer *point queries*, which retrieve records with a specified value for a search key. However, they cannot efficiently answer *range queries*, which retrieve all records whose search key value lies in a range (lb, ub). The difficulty arises because a good hash function assigns values randomly to buckets. Thus, there is no simple notion of "next bucket in sorted order." The reason we cannot chain buckets together in sorted order on A_i is that each bucket is assigned many search-key values. Since values are scattered randomly by the hash function, the values in the specified range are likely to be scattered across many or all of the buckets. Therefore, we have to read all the buckets to find the required search keys.

Recall that deletion is done as follows: If the search-key value of the record to be deleted is K_i, we compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket. Deletion of a record is efficient if there are not many records with a given key value. However, in the case of a hash index on a key with many duplicates, a large number of entries with the same key value may have to be scanned to find the entry for the record that is to be deleted. The complexity can in the worst case be linear in the number of records.

Telegram: @uni_k

www.konkur.in

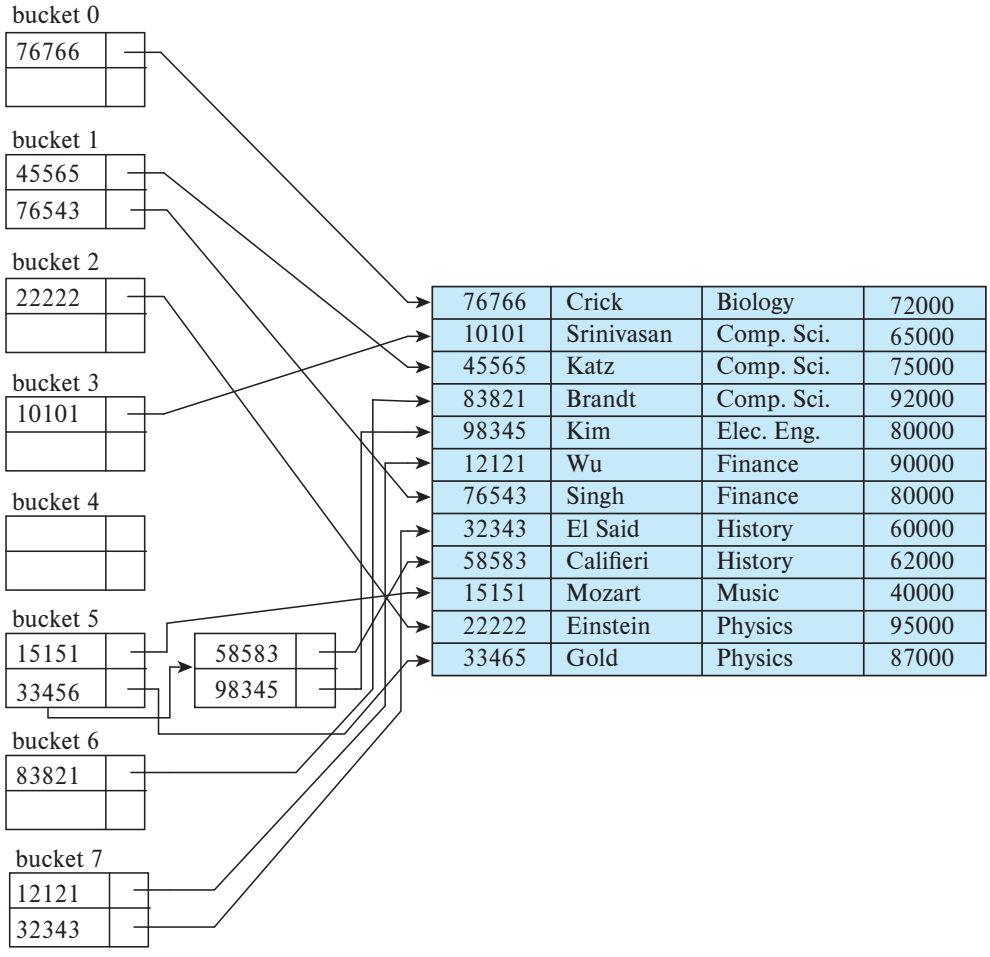


Figure 24.6 Hash index on search key *ID* of *instructor* file.

Recall also that with static hashing, the set of buckets is fixed at the time the index is created. If the relation grows far beyond the expected size, hash indices would be quite inefficient due to long overflow chains. We could rebuild the hash index using a larger number of buckets. Such rebuilding can be triggered when the number of records exceeds the estimated number by some margin, and the index is rebuilt with a number of buckets that is a multiple of the original number of buckets (say by a factor of 1.5 to 2). Such rebuilding is in fact done in many systems with in-memory hash indices.

However, doing so can cause significant disruption to normal processing with large relations, since a large number of records have to be reindexed; the disruption is particularly marked with disk-resident data. In this section we discuss dynamic hashing techniques that allow hash indices to grow gradually, without causing disruption.

Telegram: @uni_k

www.konkur.in

24.5.1.1 Hash Functions

The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired. An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.

Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:

- The distribution is *uniform*. That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.
- The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

As an illustration of these principles, let us choose a hash function for the *instructor* file using the search key *dept_name*. The hash function that we choose must have the desirable properties not only on the example *instructor* file that we have been using, but also on an *instructor* file of realistic size for a large university with many departments.

Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the *i*th letter of the alphabet to the *i*th bucket. This hash function has the virtue of simplicity, but it fails to provide a uniform distribution, since we expect more names to begin with such letters as B and R than Q and X, for example.

Now suppose that we want a hash function on the search key *salary*. Suppose that the minimum salary is $30,000 and the maximum salary is $130,000, and we use a hash function that divides the values into 10 ranges, $30,000 – $40,000, $40,001 – $50,000, and so on. The distribution of search-key values is uniform (since each bucket has the same number of different *salary* values) but is not random. Records with salaries between $60,001 and $70,000 are far more common than are records with salaries between $30,001 and $40,000. As a result, the distribution of records is not uniform —some buckets receive more records than others do. If the function has a random distribution, even if there are such correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records. (If a single search key occurs in a large fraction of the records, the bucket containing it is likely to have more records than other buckets, regardless of the hash function used.)

Telegram: @uni_k

www.konkur.in

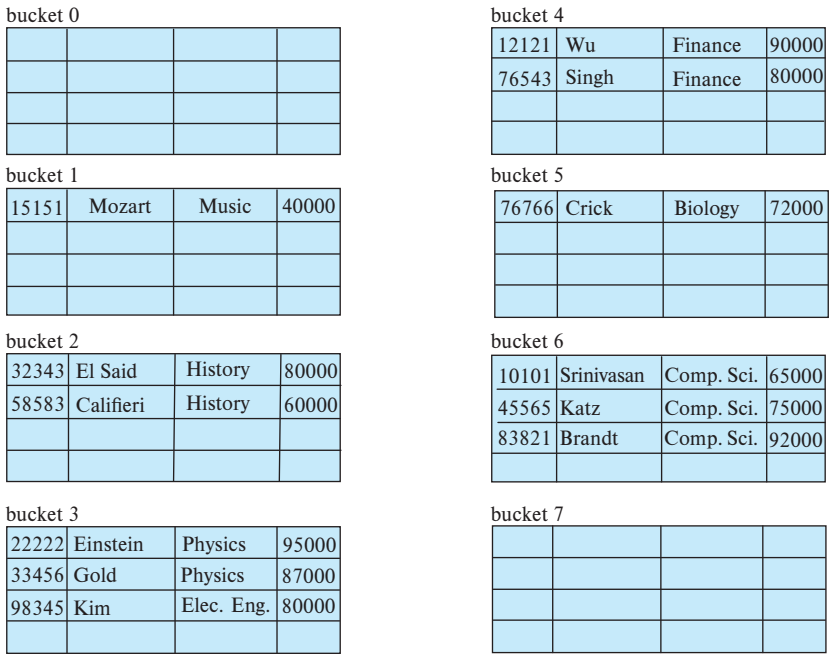


Figure 24.7 Hash organization of *instructor* file, with *dept_name* as the key.

Typical hash functions perform computation on the internal binary machine representation of characters in the search key. A simple hash function of this type first computes the sum of the binary representations of the characters of a key, then returns the sum modulo the number of buckets.

Figure 24.7 shows the application of such a scheme, with eight buckets, to the *instructor* file, under the assumption that the ith letter in the alphabet is represented by the integer i.

The following hash function is a better alternative for hashing strings. Let s be a string of length n, and let $s[i]$ denote the ith byte of the string. The hash function is defined as:

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$$

The function can be implemented efficiently by setting the hash result initially to 0 and iterating from the first to the last character of the string, at each step multiplying the hash value by 31 and then adding the next character (treated as an integer). The above expression would appear to result in a very large number, but it is actually computed with fixed-size positive integers; the result of each multiplication and addition is thus automatically computed modulo the largest possible integer value plus 1. The result of the above function modulo the number of buckets can then be used for indexing.

Telegram: @uni_k

www.konkur.in

Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

24.5.1.2 Handling of Bucket Overflows

So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons, as we outlined in Section 14.5.

- **Insufficient buckets**. The number of buckets, which we denote n_B, must be chosen such that $n_B > n_r/f_r$, where n_r denotes the total number of records that will be stored and f_r denotes the number of records that will fit in a bucket. This designation assumes that the total number of records is known when the hash function is chosen.
- **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:
  1. Multiple records may have the same search key.
  2. The chosen hash function may result in nonuniform distribution of search keys.

So that the probability of bucket overflow is reduced, the number of buckets is chosen to be $(n_r/f_r) * (1 + d)$, where d is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

Despite allocation of a few more buckets than required, bucket overflow can still occur. As we saw in Section 14.5, we handle bucket overflow by using overflow buckets. We must also change the lookup algorithm slightly to handle overflow chaining, to look at the overflow buckets in addition to the main bucket.

The form of hash structure that we have just described is called **closed addressing** (or, less commonly, **closed hashing**). Under an alternative approach called **open addressing** (or, less commonly, **open hashing**), the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets B. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used. Open addressing has been used to construct symbol tables for compilers and assemblers, but closed addressing is preferable for database systems. The reason is that deletion under open addressing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on

Telegram: @uni_k

www.konkur.in

their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open addressing is of only minor importance in database implementation.

An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function h maps search-key values to a fixed set B of bucket addresses, we waste space if B is made large to handle future growth of the file. If B is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers. We study in Section 24.5.2 how the number of buckets and the hash function can be changed dynamically.

24.5.2 Dynamic Hashing

As we have seen, the need to fix the set B of bucket addresses presents a serious problem with the static hashing technique of the previous section. Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:

1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
3. Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

Several **dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. In this section we describe one form of dynamic hashing, called **extendable hashing**. The bibliographical notes provide references to other forms of dynamic hashing.

24.5.2.1 Data Structure

Extendable hashing copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.

Telegram: @uni_k

www.konkur.in

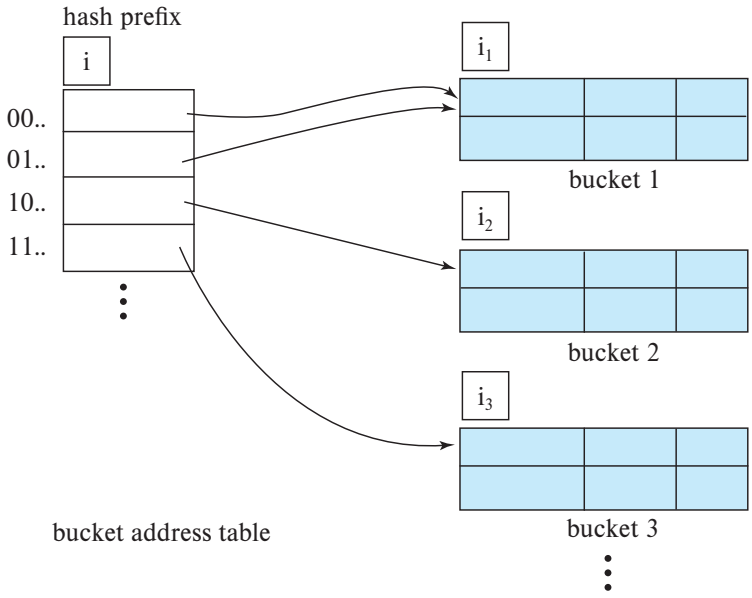


Figure 24.8 General extendable hash structure.

With extendable hashing, we choose a hash function h with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range—namely, b-bit binary integers. A typical value for b is 32.

We do not create a bucket for each hash value. Indeed, 2^{32} is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create buckets on demand, as records are inserted into the file. We do not use the entire b bits of the hash value initially. At any point, we use i bits, where $0 \leq i \leq b$. These i bits are used as an offset into an additional table of bucket addresses. The value of i grows and shrinks with the size of the database.

Figure 24.8 shows a general extendable hash structure. The i appearing above the bucket address table in the figure indicates that i bits of the hash value $h(K)$ are required to determine the correct bucket for K. This number will change as the file grows. Although i bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket. All such entries will have a common hash prefix, but the length of this prefix may be less than i. Therefore, we associate with each bucket an integer giving the length of the common hash prefix. In Figure 24.8 the integer associated with bucket j is shown as i_j. The number of bucket-address-table entries that point to bucket j is

$$2^{(i-i_j)}$$

Telegram: @uni_k

www.konkur.in

24.5.2.2 Queries and Updates

We now see how to perform lookup, insertion, and deletion on an extendable hash structure.

To locate the bucket containing search-key value K_l, the system takes the first i high-order bits of $h(K_l)$, looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.

To insert a record with search-key value K_l, the system follows the same procedure for lookup as before, ending up in some bucket–say, j. If there is room in the bucket, the system inserts the record in the bucket. If, on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.

- If $i = i_j$, only one entry in the bucket address table points to bucket j. Therefore, the system needs to increase the size of the bucket address table so that it can include pointers to the two buckets that result from splitting bucket j. It does so by considering an additional bit of the hash value. It increments the value of i by 1, thus doubling the size of the bucket address table. It replaces each entry with two entries, both of which contain the same pointer as the original entry. Now two entries in the bucket address table point to bucket j. The system allocates a new bucket (bucket z) and sets the second entry to point to the new bucket. It sets i_j and i_z to i. Next, it rehashes each record in bucket j and, depending on the first i bits (remember the system has added 1 to i), either keeps it in bucket j or allocates it to the newly created bucket.

  The system now reattempts the insertion of the new record. Usually, the attempt will succeed. However, if all the records in bucket j, as well as the new record, have the same hash-value prefix, it will be necessary to split a bucket again, since all the records in bucket j and the new record are assigned to the same bucket. If the hash function has been chosen carefully, it is unlikely that a single insertion will require that a bucket be split more than once, unless there are a large number of records with the same search key. If all the records in bucket j have the same search-key value, no amount of splitting will help. In such cases, overflow buckets are used to store the records, as in static hashing.

- If $i > i_j$, then more than one entry in the bucket address table points to bucket j. Thus, the system can split bucket j without increasing the size of the bucket address table. Observe that all the entries that point to bucket j correspond to hash prefixes that have the same value on the leftmost i_j bits. The system allocates a new bucket (bucket z), and sets i_j and i_z to the value that results from adding 1 to the original i_j value. Next, the system needs to adjust the entries in the bucket address table that previously pointed to bucket j. (Note that with the new value for i_j, not all the entries correspond to hash prefixes that have the same value on the leftmost i_j bits.) The system leaves the first half of the entries as they were (pointing to bucket

Telegram: @uni_k

www.konkur.in

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Figure 24.9 Hash function for *dept_name*.

j), and sets all the remaining entries to point to the newly created bucket (bucket z). Next, as in the previous case, the system rehashes each record in bucket j, and allocates it either to bucket j or to the newly created bucket z.

The system then reattempts the insert. In the unlikely case that it again fails, it applies one of the two cases, $i = i_j$ or $i > i_j$, as appropriate.

Note that, in both cases, the system needs to recompute the hash function on only the records in bucket j.

To delete a record with search-key value K_l, the system follows the same procedure for lookup as before, ending up in some bucket—say, j. It removes both the search key from the bucket and the record from the file. The bucket, too, is removed if it becomes empty. Note that, at this point, several buckets can be coalesced, and the size of the bucket address table can be cut in half. The procedure for deciding on which buckets can be coalesced and how to coalesce buckets is left to you to do as an exercise. The conditions under which the bucket address table can be reduced in size are also left to you as an exercise. Unlike coalescing of buckets, changing the size of the bucket address table is a rather expensive operation if the table is large. Therefore it may be worthwhile to reduce the bucket-address-table size only if the number of buckets reduces greatly.

To illustrate the operation of insertion, we use the *instructor* file and assume that the search key is *dept_name* with the 32-bit hash values as appear in Figure 24.9. Assume that, initially, the file is empty, as in Figure 24.10. We insert the records one by one. To

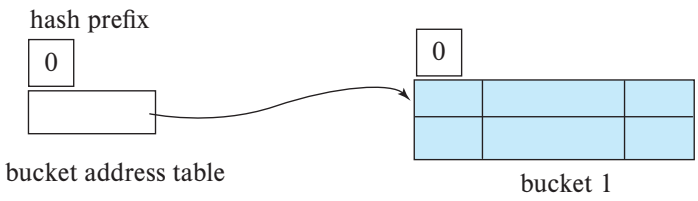


Figure 24.10 Initial extendable hash structure.

Telegram: @uni_k

www.konkur.in

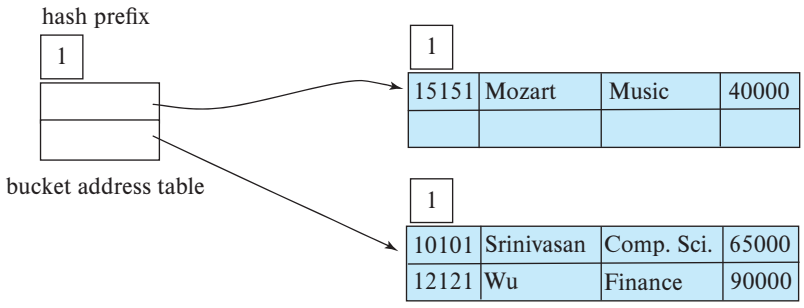


Figure 24.11 Hash structure after three insertions.

illustrate all the features of extendable hashing in a small structure, we shall make the unrealistic assumption that a bucket can hold only two records.

We insert the record (10101, Srinivasan, Comp. Sci., 65000). The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, we insert the record (12121, Wu, Finance, 90000). The system also places this record in the one bucket of our structure.

When we attempt to insert the next record (15151, Mozart, Music, 40000), we find that the bucket is full. Since $i = i_0$, we need to increase the number of bits that we use from the hash value. We now use 1 bit, allowing us $2^1 = 2$ buckets. This increase in the number of bits necessitates doubling the size of the bucket address table to two entries. The system splits the bucket, placing in the new bucket those records whose search key has a hash value beginning with 1, and leaving in the original bucket the other records. Figure 24.11 shows the state of our structure after the split.

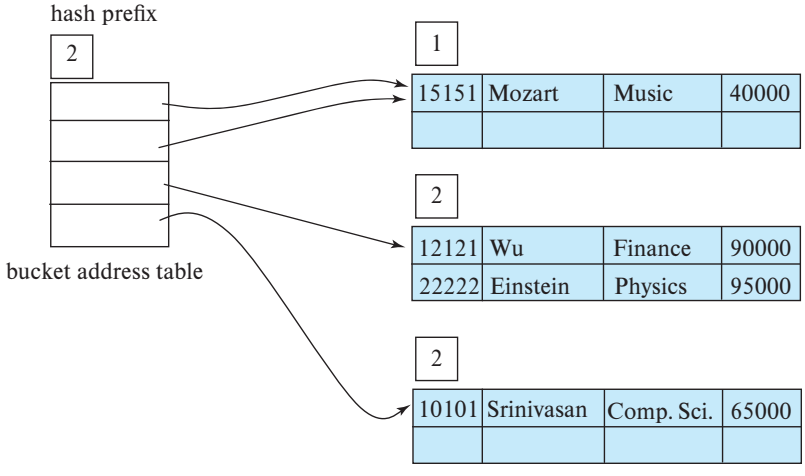


Figure 24.12 Hash structure after four insertions.

Telegram: @uni_k

www.konkur.in

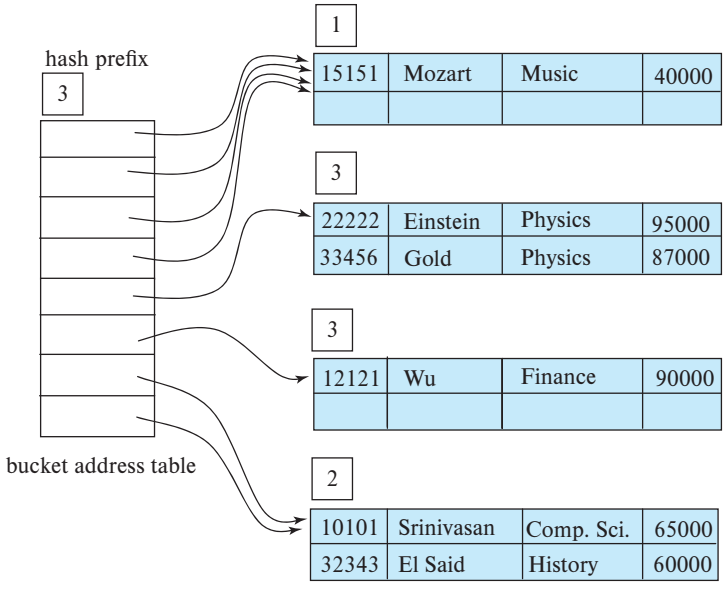


Figure 24.13 Hash structure after six insertions.

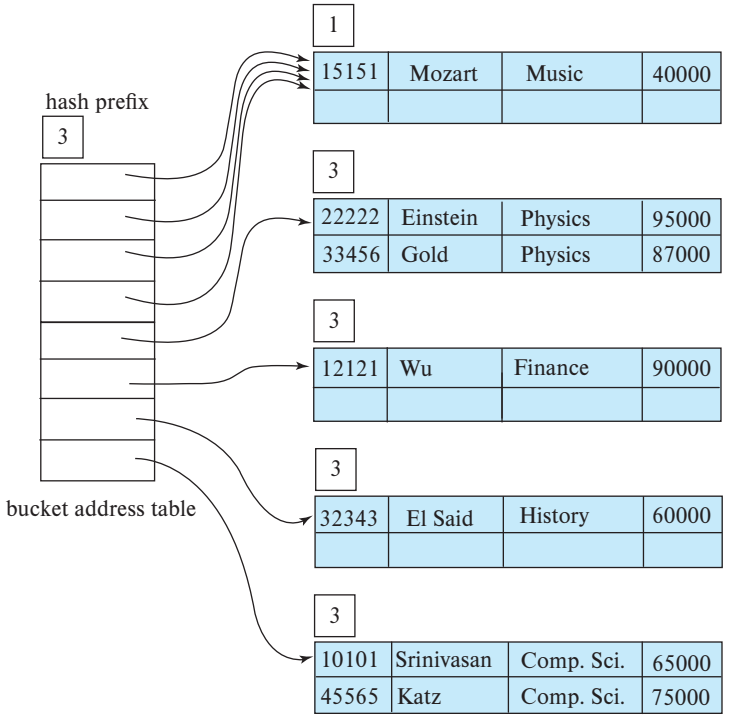


Figure 24.14 Hash structure after seven insertions.

Telegram: @uni_k

www.konkur.in

Next, we insert (22222, Einstein, Physics, 95000). Since the first bit of h(Physics) is 1, we must insert this record into the bucket pointed to by the "1" entry in the bucket address table. Once again, we find the bucket full and $i = i_1$. We increase the number of bits that we use from the hash to 2. This increase in the number of bits necessitates doubling the size of the bucket address table to four entries, as in Figure 24.12. Since the bucket of Figure 24.11 for hash prefix 0 was not split, the two entries of the bucket address table of 00 and 01 both point to this bucket.

For each record in the bucket of Figure 24.11 for hash prefix 1 (the bucket being split), the system examines the first two bits of the hash value to determine which bucket of the new structure should hold it.

Next, we insert (32343, El Said, History, 60000), which goes in the same bucket as Comp. Sci. The following insertion of (33456, Gold, Physics, 87000) results in a bucket overflow, leading to an increase in the number of bits and a doubling of the size of the bucket address table (see Figure 24.13).

The insertion of (45565, Katz, Comp. Sci., 75000) leads to another bucket overflow; this overflow, however, can be handled without increasing the number of bits, since the bucket in question has two pointers pointing to it (see Figure 24.14).

Next, we insert the records of "Califieri", "Singh", and "Crick" without any bucket overflow. The insertion of the third Comp. Sci. record (83821, Brandt, Comp. Sci.,

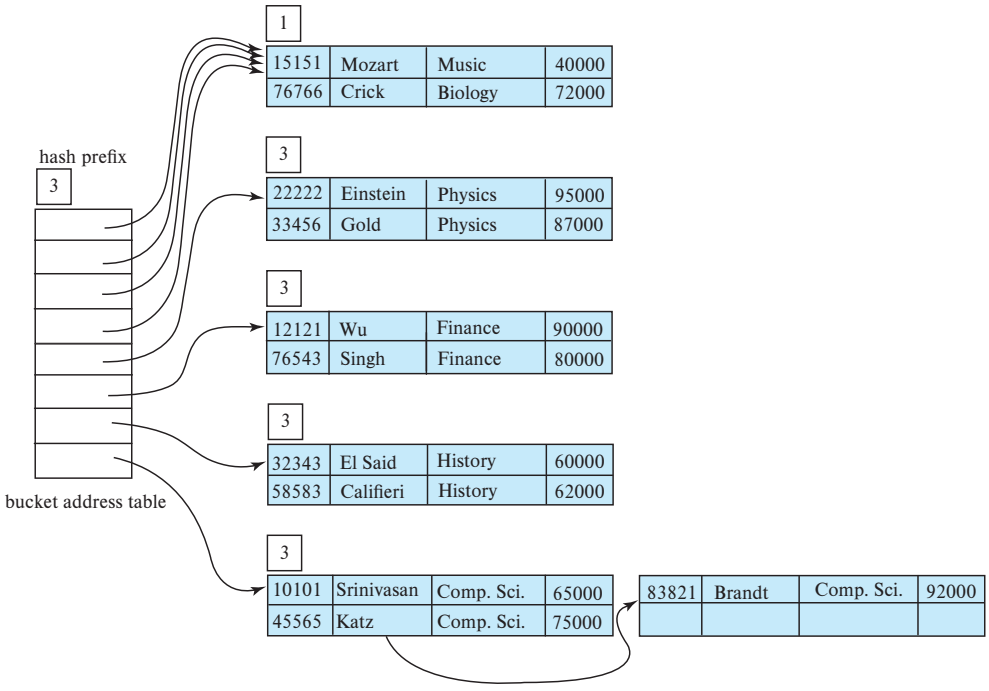


Figure 24.15 Hash structure after 11 insertions.

Telegram: @uni_k

www.konkur.in

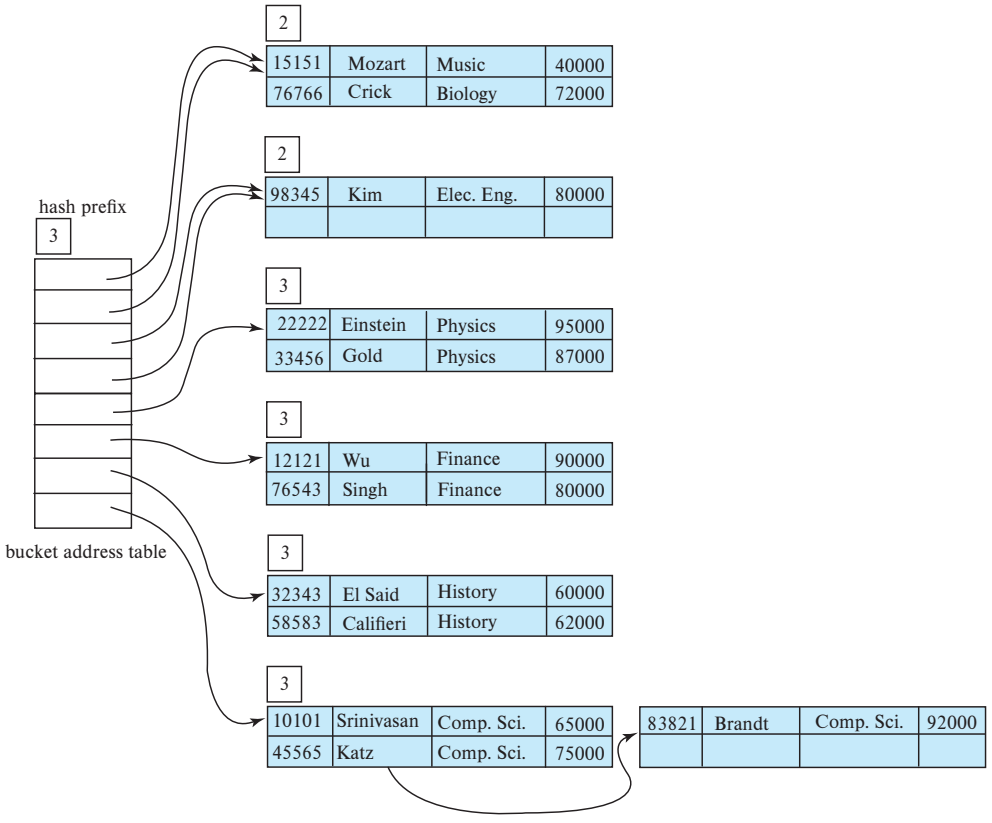


Figure 24.16 Extendable hash structure for the *instructor* file.

92000), however, leads to another overflow. This overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence the system uses an overflow bucket, as in Figure 24.15. We continue in this manner until we have inserted all the *instructor* records of Figure 14.1. The resulting structure appears in Figure 24.16.

24.5.2.3 Static Hashing versus Dynamic Hashing

We now examine the advantages and disadvantages of extendable hashing, compared with static hashing. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the current prefix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the

Telegram: @uni_k

www.konkur.in

bucket itself. This extra reference has only a minor effect on performance. Although the hash structures that we discussed in Section 24.5.1 do not have this extra level of indirection, they lose their minor performance advantage as they become full. A further disadvantage of extendable hashing is the cost of periodic doubling of the bucket address table.

The bibliographical notes also provide references to another form of dynamic hashing called **linear hashing**, which avoids the extra level of indirection associated with extendable hashing, at the possible cost of more overflow buckets.

24.5.3 Comparison of Ordered Indexing and Hashing

We have seen several ordered-indexing schemes and several hashing schemes. We can organize files of records as ordered files by using index-sequential organization or B^+-tree organizations. Alternatively, we can organize the files by using hashing. Finally, we can organize them as heap files, where the records are not ordered in any particular way.

Each scheme has advantages in certain situations. A database-system implementor could provide many schemes, leaving the final decision of which schemes to use to the database designer. However, such an approach requires the implementor to write more code, adding both to the cost of the system and to the space that the system occupies.

Most database systems support B^+-trees for indexing disk-based data, and many databases also support B^+-tree file organization. However, most databases do not support hash file organizations or hash indices for disk-based data. One of the important reasons is the fact that many applications benefit from support for range queries. A second reason is the fact that B^+-tree indices handle relation size increases gracefully, via a series of node splits, each of which is of low cost, in contrast to the relatively high cost of doubling of the bucket address table, which extendable hashing requires. Another reason for preferring B^+-trees is the fact that B^+-trees give good worst-case bounds for deletion operations with duplicate keys, unlike hash indices.

However, hash indices are used for in-memory indexing, if range queries are not common. In particular, they are widely used for creating temporary in-memory indices while processing join operations using the hash-join technique, as we see in Section 15.5.5.

24.6 Summary

- The key idea of the log structured merge tree is to replace random I/O operations during tree inserts, updates, and deletes with a smaller number of sequential I/O operations.
- Bitmap indices are specialized indices designed for easy querying on multiple keys. Bitmaps work best for attributes that take only a small number of distinct values.

Telegram: @uni_k

www.konkur.in

- A **bitmap** is an array of bits. In its simplest form, a **bitmap index** on the attribute A of relation r consists of one bitmap for each value that A can take. Each bitmap has as many bits as the number of records in the relation.
- Bitmap indices are useful for selections mainly when there are selections on multiple keys.
- An important use of bitmaps is to count the number of tuples satisfying a given selection. Such queries are important for data analysis.
- Indices are required for efficient access to spatial data and must efficiently support queries such as range and nearest neighbor queries.
- A **quadtree** is an alternative representation for two-dimensional data where the space is divided by a quadtree. Each node of a quadtree is associated with a rectangular region of space.
- An **R-tree** is a storage structure that is useful for indexing of objects such as points, line segments, rectangles, and other polygons. An R-tree is a balanced tree structure with the indexed objects stored in leaf nodes, much like a B^+-tree. However, instead of a range of values, a rectangular **bounding box** is associated with each tree node.
- **Static hashing** uses hash functions in which the set of bucket addresses is fixed. Such hash functions cannot easily accommodate databases that grow significantly larger over time.
- **Dynamic hashing techniques** allow the hash function to be modified. One example is *extendable hashing*, which copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks.

Review Terms

- Log-structured merge tree (LSM tree)
- Rolling merge
- Deletion entry
- Stepped-merge index
- Write amplification
- Bloom filter
- Bitmap
- Bitmap index
- Existence bitmap
- Quadtree
- Region quadtrees
- R-tree
- Bounding box
- Quadratic split
- Bucket overflow
- Skew
- Closed addressing
- Closed hashing
- Open addressing
- Open hashing

Telegram: @uni_k

www.konkur.in

- Dynamic hashing
- Extendable hashing
- Linear hashing

Practice Exercises

24.1 Both LSM trees and buffer trees (described in Section 14.8.2) offer benefits to write-intensive workloads, compared to normal B^+-trees, and buffer trees offer potentially better lookup performance. Yet LSM trees are more frequently used in Big Data settings. What is the most important reason for this preference?

24.2 Consider the optimized technique for counting the number of bits that are set in a bitmap. What are the tradeoffs in choosing a smaller versus a larger array size, keeping cache size in mind?

24.3 Suppose you want to store line segments in an R-tree. If a line segment is not parallel to the axes, the bounding box for it can be large, containing a large empty area.

- Describe the effect on performance of having large bounding boxes on queries that ask for line segments intersecting a given region.
- Briefly describe a technique to improve performance for such queries and give an example of its benefit. Hint: You can divide segments into smaller pieces.

24.4 Give a search algorithm on an R-tree for efficiently finding the nearest neighbor to a given query point.

24.5 Give a recursive procedure to efficiently compute the spatial join of two relations with R-tree indices. (Hint: Use bounding boxes to check if leaf entries under a pair of internal nodes may intersect.)

24.6 Suppose that we are using extendable hashing on a file that contains records with the following search-key values:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Show the extendable hash structure for this file if the hash function is $h(x) = x \bmod 8$ and buckets can hold three records.

24.7 Show how the extendable hash structure of Exercise 24.6 changes as the result of each of the following steps:

a. Delete 11.

b. Delete 31.

c. Insert 1.

Telegram: @uni_k

www.konkur.in

d. Insert 15.

24.8 Give pseudocode for deletion of entries fromAVi an extendable hash structure, including details of when and how to coalesce buckets. Do not bother about reducing the size of the bucket address table.

24.9 Suggest an efficient way to test if the bucket address table in extendable hashing can be reduced in size by storing an extra count with the bucket address table. Give details of how the count should be maintained when buckets are split, coalesced, or deleted. (*Note*: Reducing the size of the bucket address table is an expensive operation, and subsequent inserts may cause the table to grow again. Therefore, it is best not to reduce the size as soon as it is possible to do so, but instead do it only if the number of index entries becomes small compared to the bucket-address-table size.)

Exercises

24.10 The stepped merge variant of the LSM tree allows multiple trees per level. What are the tradeoffs in having more trees per level?

24.11 Suppose you want to use the idea of a quadtree for data in three dimensions. How would the resultant data structure (called an *octtree*) divide up space?

24.12 Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications.

24.13 What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?

24.14 Why is a hash structure not the best choice for a search key on which range queries are likely?

24.15 Our description of static hashing assumes that a large contiguous stretch of disk blocks can be allocated to a static hash table. Suppose you can allocate only C contiguous blocks. Suggest how to implement the hash table, if it can be much larger than C blocks. Access to a block should still be efficient.

Further Reading

The log-structured merge (LSM) tree is presented in [O'Neil et al. (1996)], while the stepped merge tree is presented in [Jagadish et al. (1997)]. [Vitter (2001)] provides an extensive survey of external-memory data structures and algorithms.

Bitmap indices are described in [O'Neil and Quass (1997)]. They were first introduced in the IBM Model 204 file manager on the AS 400 platform. They provide very

Telegram: @uni_k

www.konkur.in

large speedups on certain types of queries and are today implemented in most database systems.

[Samet (2006)] provides a textbook coverage of spatial data structures. [Bentley (1975)] describes the k-d tree, and [Robinson (1981)] describes the k-d-B tree. The R-tree was originally presented in [Guttman (1984)].

Discussions of the basic data structures in hashing can be found in [Cormen et al. (2009)]. Extendable hashing was introduced by [Fagin et al. (1979)]. Linear hashing was introduced by [Litwin (1978)] and [Litwin (1980)].

Bibliography

[Bentley (1975)] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, Volume 18, Number 9 (1975), pages 509–517.

[Cormen et al. (2009)] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition, MIT Press (2009).

[Fagin et al. (1979)] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing – A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, Volume 4, Number 3 (1979), pages 315–344.

[Guttman (1984)] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), pages 47–57.

[Jagadish et al. (1997)] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental Organization for Data Recording and Warehousing", In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97 (1997), pages 16–25.

[Litwin (1978)] W. Litwin, "Virtual Hashing: A Dynamically Changing Hashing", In *Proc. of the International Conf. on Very Large Databases* (1978), pages 517–523.

[Litwin (1980)] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", In *Proc. of the International Conf. on Very Large Databases* (1980), pages 212–223.

[O'Neil and Quass (1997)] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1997), pages 38–49.

[O'Neil et al. (1996)] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-structured Merge-tree (LSM-tree)", *Acta Inf.*, Volume 33, Number 4 (1996), pages 351–385.

[Robinson (1981)] J. Robinson, "The k-d-B Tree: A Search Structure for Large Multidimensional Indexes", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), pages 10–18.

[Samet (2006)] H. Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann (2006).

Telegram: @uni_k

www.konkur.in

[Vitter (2001)] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data", *ACM Computing Surveys*, Volume 33, (2001), pages 209–271.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

CHAPTER 25



Advanced Application Development

There are a number of tasks in application development. We saw in Chapter 6 to Chapter 9 how to design and build an application. One of the aspects of application design is the performance one expects out of the application. In fact, it is common to find that once an application has been built, it runs slower than the designers wanted or handles fewer transactions per second than they required. An application that takes an excessive amount of time to perform requested actions can cause user dissatisfaction at best and be completely unusable at worst.

Applications can be made to run significantly faster by performance tuning, which consists of finding and eliminating bottlenecks and adding appropriate hardware such as memory or disks. There are many things an application developer can do to tune the application, and there are things that a database-system administrator can do to speed up processing for an application.

Benchmarks are standardized sets of tasks that help to characterize the performance of database systems. They are useful to get a rough idea of the hardware and software requirements of an application, even before the application is built.

Applications must be tested as they are being developed. Testing requires generation of database states and test inputs, and verifying that the outputs match the expected outputs. We discuss issues in application testing. Legacy systems are application systems that are outdated and usually based on older-generation technology. However, they are often at the core of organizations and run mission-critical applications. We outline issues in interfacing with and issues in migrating away from legacy systems, replacing them with more modern systems.

Standards are very important for application development, especially in the age of the internet, since applications need to communicate with each other to perform useful tasks. A variety of standards have been proposed that affect database-application development, which we outline in this chapter. Organizations often store information about users in directory systems. Applications often use such directory systems to authenticate users and to get basic information about users, such as user categories (e.g.,

Telegram: @uni_k

www.konkur.in

student, instructor, and so on). We briefly describe the architecture of directory systems.

25.1 Performance Tuning

Tuning the performance of a system involves adjusting various parameters and design choices to improve its performance for a specific application. Various aspects of a database-system design—ranging from high-level aspects such as the schema and transaction design to database parameters such as buffer sizes, down to hardware issues such as number of disks—affect the performance of an application. Each of these aspects can be adjusted so that performance is improved.

25.1.1 Motivation for Tuning

Applications sometimes exhibit poor performance, with queries taking a long time to complete, leading to users being unable to carry out tasks that they need to do. We describe a few real-world examples that we have seen, including their causes and how tuning fixed the problems.

In one of the applications, we found that users were experiencing long delays and time-outs in the web applications. On monitoring the database, we found that the CPU usage was very high, with negligible disk and network usage. Further analysis of queries running on the database showed that a simple lookup query on a large relation was using a full relation scan, which was quite expensive. Adding an index to the attribute used in the lookup drastically reduced the execution time of the query and a key performance problem vanished immediately.

In a second application, we found that a query had very poor performance. Examining the query, we found that the programmer had written an unnecessarily complicated query, with several nested subqueries, and the optimizer produced a bad plan for the query, as we realized after observing the query plan. To fix the problem, we rewrote the query using joins instead of nested subqueries, that is, we decorrelated the query; this change greatly reduced the execution time.

In a third application, we found that the application fetched a large number of rows from a query, and issued another database query for each row that it fetched. This resulted in a large number of separate queries being sent to the database, resulting in poor performance. It is possible to replace such a large number of queries with a single query that fetches all required data, as we see later in this section. Such a change improved the performance of the application by an order of magnitude.

In a fourth application, we found that while the application performed fine under light load during testing, it completely stopped working when subjected to heavy load when it was used by actual users. In this case, we found that in some of the interfaces, programmers had forgotten to close JDBC connections. Databases typically support only a limited number of JDBC connections, and once that limit was reached, the application was unable to connect to the database, and thus it stopped working.

Telegram: @uni_k

www.konkur.in

Ensuring that connections were closed fixed this problem. While this was technically a bug fix, not a tuning action, we thought it is a good idea to highlight this problem since we have found many applications have this problem. Connection pooling, which keeps database connections open for use by subsequent transactions, is a related application tuning optimization, since it avoids the cost of repeated opening and closing of database connections.

It is also worth pointing out that in several cases above the performance problems did not show up during testing, either because the test database was much smaller than the actual database size or because the testing was done with a much lighter load (number of concurrent users) than the load on the live system. It is important that performance testing be done on realistic database sizes, with realistic load, so problems show up during testing, rather than on a live system.

25.1.2 Location of Bottlenecks

The performance of most systems (at least before they are tuned) is usually limited primarily by the performance of one or a few components, called **bottlenecks**. For instance, a program may spend 80 percent of its time in a small loop deep in the code, and the remaining 20 percent of the time on the rest of the code; the small loop then is a bottleneck. Improving the performance of a component that is not a bottleneck does little to improve the overall speed of the system; in the example, improving the speed of the rest of the code cannot lead to more than a 20 percent improvement overall, whereas improving the speed of the bottleneck loop could result in an improvement of nearly 80 percent overall, in the best case.

Hence, when tuning a system, we must first try to discover what the bottlenecks are and then eliminate them by improving the performance of system components causing the bottlenecks. When one bottleneck is removed, it may turn out that another component becomes the bottleneck. In a well-balanced system, no single component is the bottleneck. If the system contains bottlenecks, components that are not part of the bottleneck are underutilized, and could perhaps have been replaced by cheaper components with lower performance.

For simple programs, the time spent in each region of the code determines the overall execution time. However, database systems are much more complex, and query execution involves not only CPU time, but also disk I/O and network communication. A first step in diagnosing problems to use monitoring tools provided by operating systems to find the usage level of the CPU, disks, and network links.

It is also important to monitor the database itself, to find out what is happening in the database system. For example, most databases provide ways to find out which queries (or query templates, where the same query is executed repeatedly with different constants) are taking up the maximum resources, such as CPU, disk I/O, or network capacity. In addition to hardware resource bottlenecks, poor performance in a database system may potentially be due to contention on locks, where transactions wait in lock

Telegram: @uni_k

www.konkur.in



queues for a long time. Again, most databases provide mechanisms to monitor lock contention.

Monitoring tools can help detect where the bottleneck lies (such as CPU, I/O, or locks), and to locate the queries that are causing the maximum performance problems. In this chapter, we discuss a number of techniques that can be used to fix performance problems, such as adding required indices or materialized views, rewriting queries, rewriting applications, or adding hardware to improve performance.

To understand the performance of database systems better, it is very useful to model database systems as **queueing systems**. A transaction requests various services from the database system, starting from entry into a server process, disk reads during execution, CPU cycles, and locks for concurrency control. Each of these services has a queue associated with it, and small transactions may spend most of their time waiting in queues—especially in disk I/O queues—instead of executing code. Figure 25.1 illustrates some of the queues in a database system. Note that each lockable item has a separate queue in the concurrency control manager. The database system may have a single queue at the disk manager or may have separate queues for different disks in case the disks are directly controlled by the database. The transaction queue is used by the database system to control the admission of new queries when the number of requests exceeds the number of concurrent query execution tasks that the database allows.

As a result of the numerous queues in the database, bottlenecks in a database system typically show up in the form of long queues for a particular service, or, equivalently, in high utilizations for a particular service. If requests are spaced exactly uniformly, and the time to service a request is less than or equal to the time before the next request arrives, then each request will find the resource idle and can therefore start execution immediately without waiting. Unfortunately, the arrival of requests in a database system is never so uniform and is often random.

If a resource, such as a disk, has a low utilization, then when a request is made, the resource is likely to be idle, in which case the waiting time for the request will be 0. Assuming uniformly randomly distributed arrivals, the length of the queue (and

Telegram: @uni_k

www.konkur.in

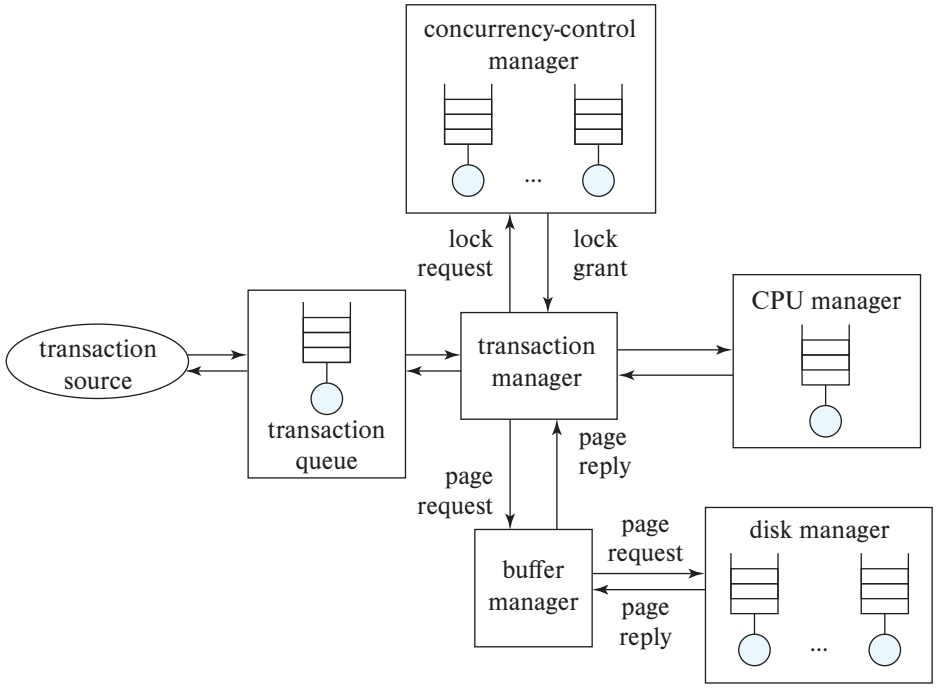


Figure 25.1 Queues in a database system.

correspondingly the waiting time) goes up exponentially with utilization; as utilization approaches 100 percent, the queue length increases sharply, resulting in excessively long waiting times. The utilization of a resource should be kept low enough that queue length is short. As a rule of the thumb, utilizations of around 70 percent are considered to be good, and utilizations above 90 percent are considered excessive, since they will result in significant delays. To learn more about the theory of queueing systems, generally referred to as **queueing theory**, you can consult the references cited in the bibliographical notes.

25.1.3 Tuning Levels

Tuning is typically done in the context of applications, and can be done at the database system layer, or outside the database system.

Tuning at layers above the database is application dependent, and is not our focus, but we mention a few such techniques. Profiling application code to find code blocks that have a heavy CPU consumption, and rewriting them to reduce CPU load is an option for CPU intensive applications. Application servers often have numerous parameters that can be tuned to improve performance, or to ensure that the application does not run out of memory. Multiple application servers that work in parallel are often

Telegram: @uni_k

www.konkur.in

used to handle higher workloads. A *load balancer* is used to route requests to one of the application servers; to ensure session continuity, requests from a particular source are always routed to the same application server. Connection pooling (described in Section 9.7.1) is another widely technique to reduce the overhead of database connection creation. Web application interfaces may be tuned to improve responsiveness, for example by replacing legacy web interfaces by ones based on JavaScript and Ajax (described in Section 9.5.1.3).

Returning to database tuning, database administrators and application developers can tune a database system at three levels.

The highest level of database tuning, which is under the control of application developers, includes the schema and queries. The developer can tune the design of the schema, the indices that are created, and the transactions that are executed to improve performance. Tuning at this level is comparatively system independent.

The second level consists of the database-system parameters, such as buffer size and checkpointing intervals. The exact set of database-system parameters that can be tuned depends on the specific database system. Most database-system manuals provide information on what database-system parameters can be adjusted, and how you should choose values for the parameters. Well-designed database systems perform as much tuning as possible automatically, freeing the user or database administrator from the burden. For instance, in many database systems the buffer size is fixed but tunable. If the system automatically adjusts the buffer size by observing indicators such as page-fault rates, then the database administrator will not have to worry about tuning the buffer size.

The lowest level is at the hardware level. Options for tuning systems at this level include replacing hard disks with solid-state drives (which use flash storage), adding more disks or using a RAID system if disk I/O is a bottleneck, adding more memory if the disk buffer size is a bottleneck, or moving to a system with more processors if CPU usage is a bottleneck.

The three levels of tuning interact with one another; we must consider them together when tuning a system. For example, tuning at a higher level may result in the hardware bottleneck changing from the disk system to the CPU, or vice versa. Tuning of queries and the physical schema is usually the first step to improving performance. Tuning of database system parameters, in case the database system does automate this task, can also be done in parallel. If performance is still poor, tuning of logical schema and tuning of hardware are the next logical steps.

25.1.4 Tuning of Physical Schema

Tuning of the physical schema, such as indices and materialized views, is the least disruptive mode of tuning, since it does not affect application code in any way. We now study different aspects of tuning of the physical schema.

Telegram: @uni_k

www.konkur.in

25.1.4.1 Tuning of Indices

We can tune the indices in a database system to improve performance. If queries are the bottleneck, we can often speed them up by creating appropriate indices on relations. If updates are the bottleneck, there may be too many indices, which have to be updated when the relations are updated. Removing indices may speed up certain updates.

The choice of the type of index also is important. Some database systems support different kinds of indices, such as hash indices, B^+-tree indices, and write-optimized indices such as LSM trees (Section 24.2). If range queries are common, B^+-tree indices are preferable to hash indices. If the system has a very high write load, but a relatively low read load, write-optimized LSM tree indices may be preferable to B^+-tree indices.

Whether to make an index a clustered index is another tunable parameter. Only one index on a relation can be made clustered, by storing the relation sorted on the index attributes. Generally, the index that benefits the greatest number of queries and updates should be made clustered.

To help identify what indices to create, and which index (if any) on each relation should be clustered, most commercial database systems provide *tuning wizards*; these are described in more detail in Section 25.1.4.4. These tools use the past history of queries and updates (called the *workload*) to estimate the effects of various indices on the execution time of the queries and updates in the workload. Recommendations on what indices to create are based on these estimates.

25.1.4.2 Using Materialized Views

Maintaining materialized views can greatly speed up certain types of queries, in particular aggregate queries. Recall the example from Section 16.5 where the total salary for each department (obtained by summing the salary of each instructor in the department) is required frequently. As we saw in that section, creating a materialized view storing the total salary for each department can greatly speed up such queries.

Materialized views should be used with care, however, since there is not only space overhead for storing them but, more important, there is also time overhead for maintaining materialized views. In the case of **immediate view maintenance**, if the updates of a transaction affect the materialized view, the materialized view must be updated as part of the same transaction. The transaction may therefore run slower. In the case of **deferred view maintenance**, the materialized view is updated later; until it is updated, the materialized view may be inconsistent with the database relations. For instance, the materialized view may be brought up to date when a query uses the view, or periodically. Using deferred maintenance reduces the burden on update transactions.

The database administrator is responsible for the selection of materialized views and for view-maintenance policies. The database administrator can make the selection manually by examining the types of queries in the workload and finding out which queries need to run faster and which updates/queries may be executed more slowly. From the examination, the database administrator may choose an appropriate set of

Telegram: @uni_k

www.konkur.in

materialized views. For instance, the administrator may find that a certain aggregate is used frequently, and choose to materialize it, or may find that a particular join is computed frequently, and choose to materialize it.

However, manual choice is tedious for even moderately large sets of query types, and making a good choice may be difficult, since it requires understanding the costs of different alternatives; only the query optimizer can estimate the costs with reasonable accuracy without actually executing the query. Thus, a good set of views may be found only by trial and error—that is, by materializing one or more views, running the workload, and measuring the time taken to run the queries in the workload. The administrator repeats the process until a set of views is found that gives acceptable performance.

A better alternative is to provide support for selecting materialized views within the database system itself, integrated with the query optimizer. This approach is described in more detail in Section 25.1.4.4.

25.1.4.3 Horizontal Partitioning of Relation Schema

Horizontal partitioning of relations is widely used for parallel and distributed storage and query processing. However, it can also be used in a centralized system to improve queries and updates by breaking up the tuples of a relation into partitions.

For example, suppose that a database stores a large relation that has a *date* attribute, and most operations work on data inserted within the past few months. Suppose now that the relation is partitioned on the *date* attribute, with one partition for each (*year, month*) combination. Then, queries that contain a selection on *date*, such as *date*='2018-06-01', need access only partitions that could possibly contain such tuples, skipping all other partitions.

More importantly, indices could be created independently on each partition. Suppose an index is created on an attribute ID, with a separate index on each partition. A query that specifies selection on ID, along with a date or a date range, need look up the index on only those partitions that match the specified date or date range. Since each partition is smaller than the whole relation, the indices too are smaller, speeding up index lookup. Index insertion is also much faster, since the index size is much smaller than an index on the entire relation. And most importantly, even as the total data size grows, the partition size never grows beyond some limit, ensuring that the performance of such queries does not degrade with time.

There is a cost to such partitioning: queries that do not contain a selection on the partitioning attribute need to individually access each of the partitions, potentially slowing down such queries significantly. If such queries are rare, the benefits of partitioning outweigh the costs, making them an attractive technique for optimization.

Even if the database does not support partitioning internally, it is possible to replace a relation r by multiple physical relations $r_1, r_2, \ldots, r_n$, and the original relation r is defined by the view $r = r_1 \cup r_2 \cup \ldots \cup r_n$. Suppose that the database optimizer knows

Telegram: @uni_k

www.konkur.in

the predicate defining each r_i (in our example, the date range corresponding to each r_i). Then the optimizer can replace a query on r that includes a selection on the partitioning attribute (*date*, in our example), with a query on the only relevant r_is. Indices would have to be created separately on each of the r_is.

25.1.4.4 Automated Tuning of Physical Design

Most commercial database systems today provide tools to help the database administrator with index and materialized view selection and other tasks related to physical database design such as how to partition data in a parallel database system.

These tools examine the **workload** (the history of queries and updates) and suggest indices and views to be materialized. The database administrator may specify the importance of speeding up different queries, which the tool takes into account when selecting views to materialize. Often tuning must be done before the application is fully developed, and the actual database contents may be small on the development database but are expected to be much larger on a production database. Thus, some tuning tools also allow the database administrator to specify information about the expected size of the database and related statistics.

Microsoft's Database Tuning Assistant, for example, allows the user to ask "what if" questions, whereby the user can pick a view, and the optimizer then estimates the effect of materializing the view on the total cost of the workload and on the individual costs of different types of queries and updates in the workload.

The automatic selection of indices and materialized views is usually implemented by enumerating different alternatives and using the query optimizer to estimate the costs and benefits of selecting each alternative by using the workload. Since the number of design alternatives and the potential workload may be extremely large, the selection techniques must be designed carefully.

The first step is to generate a workload. This is usually done by recording all the queries and updates that are executed during some time period. Next, the selection tools perform **workload compression**, that is, create a representation of the workload using a small number of updates and queries. For example, updates of the same form can be represented by a single update with a weight corresponding to how many times the update occurred. Queries of the same form can be similarly replaced by a representative with appropriate weight. After this, queries that are very infrequent and do not have a high cost may be discarded from consideration. The most expensive queries may be chosen to be addressed first. Such workload compression is essential for large workloads.

With the help of the optimizer, the tool would come up with a set of indices and materialized views that could help the queries and updates in the compressed workload. Different combinations of these indices and materialized views can be tried out to find the best combination. However, an exhaustive approach would be totally impractical, since the number of potential indices and materialized views is already large, and each

Telegram: @uni_k

www.konkur.in



subset of these is a potential design alternative, leading to an exponential number of alternatives. Heuristics are used to reduce the space of alternatives, that is, to reduce the number of combinations considered.

Greedy heuristics for index and materialized view selection operate as follows: They estimate the benefits of materializing different indices or views (using the optimizer's cost estimation functionality as a subroutine). They then choose the index or view that gives either the maximum benefit or the maximum benefit per unit space (i.e., benefit divided by the space required to store the index or view). The cost of maintaining the index or view must be taken into account when computing the benefit. Once the heuristic has selected an index or view, the benefits of other indices or views may have changed, so the heuristic recomputes these and chooses the next best index or view for materialization. The process continues until either the available disk space for storing indices or materialized views is exhausted or the cost of maintaining the remaining candidates is more than the benefit to queries that could use the indices or views.

Real-world index and materialized-view selection tools usually incorporate some elements of greedy selection but use other techniques to get better results. They also support other aspects of physical database design, such as deciding how to partition a relation in a parallel database, or what physical storage mechanism to use for a relation.

Telegram: @uni_k

www.konkur.in

25.1.5 Tuning of Queries

The performance of an application can often be significantly improved by rewriting queries or by changing how the application issues queries to the database.

25.1.5.1 Tuning of Query Plans

In the past, optimizers on many database systems were not particularly good, so how a query was written would have a big influence on how it was executed, and therefore on the performance. Today's advanced optimizers can transform even badly written queries and execute them efficiently, so the need for tuning individual queries is less important than it used to be. However, sometimes query optimizers choose bad plans for one of several reasons, which we describe next.

Before checking if something needs to be tuned in the plan for a query, it is useful to find out what plan is being used for the query. Most databases support an explain command, which allows you to see what plan is being used for a query. The explain command also shows the statistics that the optimizer used or computed for different parts of the query plan, and estimates of the costs of each part of a query plan. Variants of the explain command also execute the query and get actual tuple counts and execution time for different parts of the query plan.

Incorrect statistics are often the reason for the choice of a bad plan. For example, if the optimizer thinks that the relations involved in a join have very few tuples, it may choose nested loops join, which would be very inefficient if the relations actually have a large number of tuples.

Ideally, database statistics should be updated whenever relations are updated. However, doing so adds unacceptable overhead to update queries. Instead, databases either periodically update statistics or leave it to the system administrator to issue a command to update statistics. Some databases, such as PostgreSQL and MySQL support a command called analyze,[1] which can be used to recompute statistics. For example, analyze instructor would recompute statistics for the instructor relation, while analyze with no arguments would recompute statistics for all relations in PostgreSQL. It is highly recommended to run this command after loading data into the database, or after making a significant number of inserts or deletes on a relation.

Some databases such as Oracle and Microsoft SQL Server keep track of inserts and deletes to relations, and they update statistics whenever the relation size changes by a significant fraction, making execution of the analyze command unnecessary.

Another reason for poor performance of queries is the lack of required indices. As we saw earlier, the choice of indices can be done as part of the tuning of the physical schema, but examining a query helps us understand what indices may be useful to speed up that query.

Indices are particularly important for queries that fetch only a few rows from a large relation, based on a predicate. For example, a query that finds students in a department

[1] The command is called analyze table in the case of MySQL.

Telegram: @uni_k

www.konkur.in

may benefit from an index on the *student* relation on the attribute *dept_name*. Indices on join attributes are often very useful. For example, if the above query also included a join of *student* with *takes* on the attribute *takes*.ID, an index on *takes*.ID could be useful.

Note that databases typically create indices on primary-key attributes, which can be used for selections as well as joins. For example, in our university schema, the primary-key index on *takes* has ID as its first attribute and may thus be useful for the above join.

Complex queries containing *nested subqueries* are not optimized very well by many optimizers. We saw techniques for nested subquery decorrelation in Section 16.4.4. If a subquery is not decorrelated, it gets executed repeatedly, potentially resulting in a great deal of random I/O. In contrast, decorrelation allows efficient set-oriented operations such as joins to be used, minimizing random I/O. Most database query optimizers incorporate some forms of decorrelation, but some can handle only very simple nested subqueries. The execution plan chosen by the optimizer can be found as described in Chapter 16. If the optimizer has not succeeded in decorrelating a nested subquery, the query can be decorrelated by rewriting it manually.

25.1.5.2 Improving Set Orientation

When SQL queries are executed from an application program, it is often the case that a query is executed frequently, but with different values for a parameter. Each call has an overhead of communication with the server, in addition to processing overheads at the server.

For example, consider a program that steps through each department, invoking an embedded SQL query to find the total salary of all instructors in the department:

```
select sum(salary)
from instructor
where dept_name= ?
```

If the *instructor* relation does not have a clustered index on *dept_name*, each such query will result in a scan of the relation. Even if there is such an index, a random I/O operation will be required for each *dept_name* value.

Instead, we can use a single SQL query to find total salary expenses of each department:

```
select dept_name, sum(salary)
from instructor
group by dept_name;
```

This query can be evaluated with a single scan of the *instructor* relation, avoiding random I/O for each department. The results can be fetched to the client side using a single round of communication, and the client program can then step through the results to find the aggregate for each department. Combining

Telegram: @uni_k

www.konkur.in


```
PreparedStatement pStmt = conn.prepareStatement(
                "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setInt(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.addBatch( );
pStmt.setString(1, "88878");
pStmt.setString(2, "Thierry");
pStmt.setInt(3, "Physics");
pStmt.setInt(4, 100000);
pStmt.addBatch( );
pStmt.executeBatch( );
```

Figure 25.2 Batch update in JDBC.

SQL query as above can reduce execution costs greatly in many cases—for example, if the *instructor* relation is very large and has a large number of departments.

The JDBC API also provides a feature called **batch update** that allows a number of inserts to be performed using a single communication with the database. Figure 25.2 illustrates the use of this feature. The code shown in the figure requires only one round of communication with the database, when the executeBatch() method is executed, in contrast to similar code without the batch update feature that we saw in Figure 5.2. In the absence of batch update, as many rounds of communication with the database are required as there are instructors to be inserted. The batch update feature also enables the database to process a batch of inserts at once, which can potentially be done much more efficiently than a series of single record inserts.

Another technique used widely in client-server systems to reduce the cost of communication and SQL compilation is to use stored procedures, where queries are stored at the server in the form of procedures, which may be precompiled. Clients can invoke these stored procedures rather than communicate a series of queries.

25.1.5.3 Tuning of Bulk Loads and Updates

When loading a large volume of data into a database (called a **bulk load** operation), performance is usually very poor if the inserts are carried out as separate SQL insert statements. One reason is the overhead of parsing each SQL query; a more important reason is that performing integrity constraint checks and index updates separately for each inserted tuple results in a large number of random I/O operations. If the inserts were done as a large batch, integrity-constraint checking and index update can be done

Telegram: @uni_k

www.konkur.in

in a much more set-oriented fashion, reducing overheads greatly; performance improvements of an order of magnitude or more are not uncommon.

To support bulk load operations, most database systems provide a **bulk import** utility and a corresponding **bulk export** utility. The bulk-import utility reads data from a file and performs integrity constraint checking as well as index maintenance in a very efficient manner. Common input and output file formats supported by such bulk import/export utilities include text files with characters such as commas or tabs separating attribute values, with each record in a line of its own (such file formats are referred to as *comma-separated values* or *tab-separated values* formats). Database-specific binary formats as well as XML formats are also supported by bulk import/export utilities. The names of the bulk import/export utilities differ by database. In PostgreSQL, the utilities are called pg_dump and pg_restore (PostgreSQL also provides an SQL command **copy**, which provides similar functionality). The bulk import/export utility in Oracle is called **SQL*Loader**, the utility in DB2 is called load, and the utility in SQL Server is called bcp (SQL Server also provides an SQL command called **bulk insert**).

We now consider the case of tuning of bulk updates. Suppose we have a relation *funds_received*(*dept_name*, *amount*) that stores funds received (say, by electronic funds transfer) for each of a set of departments. Suppose now that we want to add the amounts to the balances of the corresponding department budgets. In order to use the SQL update statement to carry out this task, we have to perform a look up on the *funds_received* relation for each tuple in the *department* relation. We can use subqueries in the update clause to carry out this task, as follows: We assume for simplicity that the relation *funds_received* contains at most one tuple for each department.

```
update department set budget = budget +
                    (select amount
                     from funds_received
                     where funds_received.dept_name = department.dept_name)
        where exists(
                    select *
                     from funds_received
                     where funds_received.dept_name = department.dept_name);
```

Note that the condition in the **where** clause of the update ensures that only accounts with corresponding tuples in *funds_received* are updated, while the subquery within the **set** clause computes the amount to be added to each such department.

There are many applications that require updates such as that illustrated above. Typically, there is a table, which we shall call the **master table**, and updates to the master table are received as a batch. Now the master table has to be correspondingly updated. SQL:2003 introduced a special construct, called the **merge** construct, to simplify the task of performing such merging of information. For example, the preceding update can be expressed using **merge** as follows:

Telegram: @uni_k

www.konkur.in


```
merge into department as A
using       (select *
             from funds_received) as F
      on (A.dept_name = F.dept_name)
when matched then
      update set budget = budget + F.amount;
```

When a record from the subquery in the **using** clause matches a record in the *department* relation, the **when matched** clause is executed, which can execute an update on the relation; in this case, the matching record in the *department* relation is updated as shown.

The **merge** statement can also have a **when not matched then** clause, which permits insertion of new records into the relation. In the preceding example, when there is no matching department for a *funds_received* tuple, the insertion action could create a new department record (with a null *building*) using the following clause:

```
when not matched then
      insert values (F.dept_name, null, F.budget)
```

Although not very meaningful in this example,[2] the **when not matched then** clause can be quite useful in other cases. For example, suppose the local relation is a copy of a master relation, and we receive updated as well as newly inserted records from the master relation. The **merge** statement can update matched records (these would be updated old records) and insert records that are not matched (these would be new records).

Not all SQL implementations support the **merge** statement currently; see the respective system manuals for further details.

25.1.6 Tuning of the Logical Schema

Performance of queries can sometimes be improved by tuning of the logical schema. For example, within the constraints of the chosen normal form, it is possible to partition relations vertically. Consider the *course* relation, with the schema:

course (*course_id*, *title*, *dept_name*, *credits*)

for which *course_id* is a key. Within the constraints of the normal forms (BCNF and 3NF), we can partition the *course* relation into two relations:

course_credit (*course_id*, *credits*)
course_title_dept (*course_id*, *title*, *dept_name*)

[2] A better action here would have been to insert these records into an error relation, but that cannot be done with the **merge** statement.

Telegram: @uni_k

www.konkur.in

The two representations are logically equivalent, since *course_id* is a key, but they have different performance characteristics.

If most accesses to course information look at only the *course_id* and *credits*, then they can be run against the *course_credit* relation, and access is likely to be somewhat faster, since the *title* and *dept_name* attributes are not fetched. For the same reason, more tuples of *course_credit* will fit in the buffer than corresponding tuples of *course*, again leading to faster performance. This effect would be particularly marked if the *title* and *dept_name* attributes were large. Hence, a schema consisting of *course_credit* and *course_title_dept* would be preferable to a schema consisting of the *course* relation in this case.

On the other hand, if most accesses to course information require both *dept_name* and *credits*, using the *course* relation would be preferable, since the cost of the join of *course_credit* and *course_title_dept* would be avoided. Also, the storage overhead would be lower, since there would be only one relation, and the attribute *course_id* would not be replicated.

The **column store** approach to storing data are based on vertical partitioning but takes it to the limit by storing each attribute (column) of the relation in a separate file, as we saw in Section 13.6. Note that in a column store it is not necessary to repeat the primary-key attribute since the i^{th} row can be reconstructed by taking the i^{th} entry for each desired column. Column stores have been shown to perform well for several data-warehouse applications by reducing I/O, improving cache performance, enabling greater gains from data compression, and allowing effective use of CPU vector-processing capabilities.

Another trick to improve performance is to store a *denormalized relation*, such as a join of *instructor* and *department*, where the information about *dept_name*, *building*, and *budget* is repeated for every instructor. More effort has to be expended to make sure the relation is consistent whenever an update is carried out. However, a query that fetches the names of the instructors and the associated buildings will be speeded up, since the join of *instructor* and *department* will have been precomputed. If such a query is executed frequently, and has to be performed as efficiently as possible, the denormalized relation could be beneficial.

Materialized views can provide the benefits that denormalized relations provide, at the cost of some extra storage. A major advantage to materialized views over denormalized relations is that maintaining consistency of redundant data becomes the job of the database system, not the programmer. Thus, materialized views are preferable, whenever they are supported by the database system.

Another approach to speed up the computation of the join without materializing it is to cluster records that would match in the join on the same disk page. We saw such clustered file organizations in Section 13.3.3.

25.1.7 Tuning of Concurrent Transactions

Concurrent execution of different types of transactions can sometimes lead to poor performance because of contention on locks. We first

Telegram: @uni_k

www.konkur.in

contention, which is more common, and then consider the case of write-write contention.

As an example of **read-write contention**, consider the following situation on a banking database. During the day, numerous small update transactions are executed almost continuously. Suppose that a large query that computes statistics on branches is run at the same time. If the query performs a scan on a relation, it may block out all updates on the relation while it runs, and that can have a disastrous effect on the performance of the system.

Several database systems—Oracle, PostgreSQL, and Microsoft SQL Server, for example— support snapshot isolation, whereby queries are executed on a snapshot of the data, and updates can go on concurrently. (Snapshot isolation is described in detail in Section 18.8.) Snapshot isolation should be used, if available, for large queries, to avoid lock contention in the above situation. In SQL Server, executing the statement

set transaction isolation level snapshot

at the beginning of a transaction results in snapshot isolation being used for that transaction. In Oracle and PostgreSQL, using the keyword **serializable** in place of the keyword **snapshot** in the above command has the same effect, since these systems actually use snapshot isolation (serializable snapshot isolation, in the case of PostgreSQL version 9.1 onwards) when the isolation level is set to serializable.

If snapshot isolation is not available, an alternative option is to execute large queries at times when updates are few or nonexistent. However, for databases supporting web sites, there may be no such quiet period for updates.

Another alternative is to use weaker levels of consistency, such as the **read committed** isolation level, whereby evaluation of the query has a minimal impact on concurrent updates, but the query results are not guaranteed to be consistent. The application semantics determine whether approximate (inconsistent) answers are acceptable.

We now consider the case of **write-write contention**. Data items that are updated very frequently can result in poor performance with locking, with many transactions waiting for locks on those data items. Such data items are called **update hot spots**. Update hot spots can cause problems even with snapshot isolation, causing frequent transaction aborts due to write-validation failures. A commonly occurring situation that results in an update hot spot is as follows: transactions need to assign unique identifiers to data items being inserted into the database, and to do so they read and increment a sequence counter stored in a tuple in the database. If inserts are frequent, and the sequence counter is locked in a two-phase manner, the tuple containing the sequence counter becomes a hot spot.

One way to improve concurrency is to release the lock on the sequence counter immediately after it is read and incremented; however, after doing so, even if the transaction aborts, the update to the sequence counter should not be rolled back. To understand why, suppose T_1 increments the sequence counter, and then T_2 increments the sequence counter before T_1 commits; if T_1 then aborts, rolling back its update, either

Telegram: @uni_k

www.konkur.in

by restoring the counter to the original value or by decrementing the counter, will result in the sequence value used by T_2 getting reused by a subsequent transaction.

Most databases provide a special construct for creating **sequence counters** that implement early, non-two-phase lock release, coupled with special-case treatment of undo logging so that updates to the counter are not rolled back if the transaction aborts. The SQL standard allows a sequence counter to be created using the command:

create sequence *counter1*;

In the above command, *counter1* is the name of the sequence; multiple sequences can be created with different names. The syntax to get a value from the sequence is not standardized; in Oracle, *counter1.nextval* would return the next value of the sequence, after incrementing it, while the function call *nextval* ('*counter1*') would have the same effect in PostgreSQL, and DB2 uses the syntax **nextval for** *counter1*.

The SQL standard provides an alternative to using an explicit sequence counter, which is useful when the goal is to give unique identifiers to tuples inserted into a relation. To do so, the keyword **identity** can be added to the declaration of an integer attribute of a relation (usually this attribute would also be declared as the primary key). If the value for that attribute is left unspecified in an insert statement for that relation, a unique new value is created automatically for each newly inserted tuple. A non-two-phase locked sequence counter is used internally to implement the **identity** declaration, with the counter incremented each time a tuple is inserted. Several databases, including DB2 and SQL Server support the **identity** declaration, although the syntax varies. PostgreSQL supports a data type called **serial**, which provides the same effect; the PostgreSQL type **serial** is implemented by transparently creating a non-two-phase locked sequence.

It is worth noting that since the acquisition of a sequence number by a transaction cannot be rolled back if the transaction aborts (for reasons discussed earlier), transaction aborts may result in *gaps in the sequence numbers* in tuples inserted in the database. For example, there may be tuples with identifier value 1001 and 1003, but no tuple with value 1002, if the transaction that acquired the sequence number 1002 did not commit. Such gaps are not acceptable in some applications; for example, some financial applications require that there be no gaps in bill or receipt numbers. Database provided sequences and automatically incremented attributes should not be used for such applications, since they can result in gaps. A sequence counter stored in normal tuples, which is locked in a two-phase manner, would not be susceptible to such gaps since a transaction abort would restore the sequence counter value, and the next transaction would get the same sequence number, avoiding a gap.

Long update transactions can cause performance problems with system logs and can increase the time taken to recover from system crashes. If a transaction performs many updates, the system log may become full even before the transaction completes, in which case the transaction will have to be rolled back. If an update transaction runs for a long time (even with few updates), it may block deletion of old parts of the log,

Telegram: @uni_k

www.konkur.in

if the logging system is not well designed. Again, this blocking could lead to the log getting filled up.

To avoid such problems, many database systems impose strict limits on the number of updates that a single transaction can carry out. Even if the system does not impose such limits, it is often helpful to break up a large update transaction into a set of smaller update transactions where possible. For example, a transaction that gives a raise to every employee in a large corporation could be split up into a series of small transactions, each of which updates a small range of employee-ids. Such transactions are called **minibatch transactions**. However, minibatch transactions must be used with care. First, if there are concurrent updates on the set of employees, the result of the set of smaller transactions may not be equivalent to that of the single large transaction. Second, if there is a failure, the salaries of some of the employees would have been increased by committed transactions, but salaries of other employees would not. To avoid this problem, as soon as the system recovers from failure, we must execute the transactions remaining in the batch.

Long transactions, whether read-only or update, can also result in the lock table becoming full. If a single query scans a large relation, the query optimizer would ensure that a relation lock is obtained instead of acquiring a large number of tuple locks. However, if a transaction executes a large number of small queries or updates, it may acquire a large number of locks, resulting in the lock table becoming full.

To avoid this problem, some databases provide for automatic **lock escalation**; with this technique, if a transaction has acquired a large number of tuple locks, tuple locks are upgraded to page locks, or even full relation locks. Recall that with multiple-granularity locking (Section 18.3), once a coarser-level lock is obtained, there is no need to record finer-level locks, so tuple lock entries can be removed from the lock table, freeing up space. On databases that do not support lock escalation, it is possible for the transaction to explicitly acquire a relation lock, thereby avoiding the acquisition of tuple locks.

25.1.8 Tuning of Hardware

Hardware bottlenecks could include memory, I/O, CPU and network capacity. We focus on memory and I/O tuning in this section. The availability of processors with a large number of CPU cores, and support for multiple CPUs on a single machine allows system designers to choose the CPU model and number of CPUs to meet the CPU requirements of the application at an acceptable cost. How to tune or choose between CPU and network interconnect options is a topic outside the domain of database tuning.

Even in a well-designed transaction processing system, each transaction usually has to do at least a few I/O operations, if the data required by the transaction are on disk. An important factor in tuning a transaction processing system is to make sure that the disk subsystem can handle the rate at which I/O operations are required. For instance, consider a hard disk that supports an access time of about 10 milliseconds, and average transfer rate of 25 to 100 megabytes per second (a fairly typical disk today). Such a disk

Telegram: @uni_k

www.konkur.in

would support a little under 100 random-access I/O operations of 4 kilobytes each per second. If each transaction requires just two I/O operations, a single disk would support at most 50 transactions per second.

An obvious way to improve performance is to replace a hard disk with a solid-state drive (SSD), since a single SSD can support tens of thousands of random I/O operations per second. A drawback of using SSDs is that they cost a lot more than hard disks for a given storage capacity. Another way to support more transactions per second is to increase the number of disks. If the system needs to support n transactions per second, each performing two I/O operations, data must be striped (or otherwise partitioned) across at least $n/50$ hard disks (ignoring skew), or $n/5000$ SSDs, if the SSD supports 10,000 random I/O operations per second.

Notice here that the limiting factor is not the capacity of the disk, but the speed at which random data can be accessed (limited in a hard disk by the speed at which the disk arm can move). The number of I/O operations per transaction can be reduced by storing more data in memory. If all data are in memory, there will be no disk I/O except for writes. Keeping frequently used data in memory reduces the number of disk I/Os and is worth the extra cost of memory. Keeping very infrequently used data in memory would be a waste, since memory is much more expensive than disk.

The question is, for a given amount of money available for spending on disks or memory, what is the best way to spend the money to achieve the maximum number of transactions per second? A reduction of one I/O per second saves:

$$(\textit{price per disk drive})/(\textit{access per second per disk})$$

Thus, if a particular page is accessed once in m seconds, the saving due to keeping it in memory is $\frac{1}{m}$ times the above value. Storing a page in memory costs:

$$(\textit{price per megabyte of memory})/(\textit{pages per megabyte of memory})$$

Thus, the break-even point is:

$$\frac{1}{m} * \frac{\textit{price per disk drive}}{\textit{access per second per disk}} = \frac{\textit{price per megabyte of memory}}{\textit{pages per megabyte of memory}}$$

We can rearrange the equation and substitute current values for each of the above parameters to get a value for m; if a page is accessed more frequently than once in m seconds, it is worth buying enough memory to store it.

As of 2018, hard-disk technology and memory and disk prices (which we assume to be about $50 for a 1-terabyte disk and $80 for 16-gigabytes of memory) give a value of m around 4 hours for 4-kilobytes pages that are randomly accessed; that is, if a page on hard disk is accessed at least once in 4 hours, it makes sense to purchase enough memory to cache it in memory. Note that if we use larger pages, the time decreases; for example, a page size of 16-kilobytes will lead to a value of m of 1 hour instead of 4 hours.

Telegram: @uni_k

www.konkur.in

With disk and memory cost and speeds as of the 1980/1990s, the corresponding value was 5 minutes with 4-kilobytes pages. Thus, a widely used rule of thumb, called the **five minute rule**, which said that data should be cached in memory if it is accessed more frequently than once in 5 minutes.

With SSD technology and prices as of 2018 (which we assume to be around $500 for a 800 gigabytes SSD, which supports 67,000 random reads and 20,000 random writes per second), if we make the same comparison between keeping a page in memory versus fetching it from SSD, the time comes to around 7 minutes with 4-kilobyte pages. That is, if a page on SSD is accessed more frequently than once in 7 minutes, it is worth purchasing enough memory to cache it in memory.

For data that are sequentially accessed, significantly more pages can be read per second. Assuming 1 megabyte of data are read at a time, the breakeven point for hard disk currently is about 2.5 minutes. Thus, sequentially accessed data on hard disk should be cached in memory if they are used at least once in 2.5 minutes. For SSDs, the breakeven point is much smaller, at 1.6 seconds. In other words, there is little benefit in caching sequentially accessed data in memory unless it is very frequently accessed.

The above rules of thumb take only the number of I/O operations per second into account and do not consider factors such as response time. Some applications need to keep even infrequently used data in memory to support response times that are less than or comparable to disk-access time.

Since SSD storage is more expensive than disk, one way to get faster random I/O for frequently used data, while paying less for storing less frequently used data, is to use the **flash-as-buffer** approach. In this approach, flash storage is used as a persistent buffer, with each block having a permanent location on disk, but stored in flash instead of being written to disk as long as it is frequently used. When flash storage is full, a block that is not frequently used is evicted and flushed back to disk if it was updated after being read from disk. Disk subsystems that provide hard disks along with SSDs that act as buffers are commercially available. A rule of thumb for deciding how much SSD storage to purchase is that a 4-kilobyte page should be kept on SSD, instead of hard disk, if it is accessed more frequently than once in a day (the computation is similar to the case of caching in main memory versus fetching from disk/SSD). Note that in such a setup, the database system cannot control what data reside in which part of the storage.

If the storage system allows direct access to SSDs as well as hard disks, the database administrator can control the mapping of relations or indices to disks and allocate frequently used relations/indices to flash storage. The tablespace feature, supported by most database systems, can be used to control the mapping by creating a tablespace on flash storage and assigning desired relations and indices to that tablespace. Controlling the mapping at a finer level of granularity than a relation, however, requires changes to the database-system code.

Another aspect of tuning is whether to use RAID 1 or RAID 5. The answer depends on how frequently the data are updated, since RAID 5 is much slower than RAID 1 on

Telegram: @uni_k

www.konkur.in

random writes: RAID 5 requires 2 reads and 2 writes to execute a single random write request. If an application performs r random reads and w random writes per second to support a particular throughput rate, a RAID 5 implementation would require $r + 4w$ I/O operations per second, whereas a RAID 1 implementation would require $r + 2w$ I/O operations per second. We can then calculate the number of disks required to support the required I/O operations per second by dividing the result of the calculation by 100 I/O operations per second (for current-generation disks). For many applications, r and w are large enough that the $(r + w)/100$ disks can easily hold two copies of all the data. For such applications, if RAID 1 is used, the required number of disks is actually less than the required number of disks if RAID 5 is used! Thus, RAID 5 is useful only when the data storage requirements are very large, but the update rates, and particularly random update rates, are small.

25.1.9 Performance Simulation

To test the performance of a database system even before it is installed, we can create a performance-simulation model of the database system. Each service shown in Figure 25.1, such as the CPU, each disk, the buffer, and the concurrency control, is modeled in the simulation. Instead of modeling details of a service, the simulation model may capture only some aspects of each service, such as the **service time**—that is, the time taken to finish processing a request once processing has begun. Thus, the simulation can model a disk access from just the average disk-access time.

Since requests for a service generally have to wait their turn, each service has an associated queue in the simulation model. A transaction consists of a series of requests. The requests are queued up as they arrive and are serviced according to the policy for that service, such as first come, first served. The models for services such as CPU and the disks conceptually operate in parallel, to account for the fact that these subsystems operate in parallel in a real system.

Once the simulation model for transaction processing is built, the system administrator can run a number of experiments on it. The administrator can use experiments with simulated transactions arriving at different rates to find how the system would behave under various load conditions. The administrator could run other experiments that vary the service times for each service to find out how sensitive the performance is to each of them. System parameters, too, can be varied, so that performance tuning can be done on the simulation model.

25.2 Performance Benchmarks

As database servers become more standardized, the differentiating factor among the products of different vendors is those products' performance. **Performance benchmarks** are suites of tasks that are used to quantify the performance of software systems.

Telegram: @uni_k

www.konkur.in

25.2.1 Suites of Tasks

Since most software systems, such as databases, are complex, there is a good deal of variation in their implementation by different vendors. As a result, there is a significant amount of variation in their performance on different tasks. One system may be the most efficient on a particular task; another may be the most efficient on a different task. Hence, a single task is usually insufficient to quantify the performance of the system. Instead, the performance of a system is measured by suites of standardized tasks, called *performance benchmarks*.

Combining the performance numbers from multiple tasks must be done with care. Suppose that we have two tasks, T_1 and T_2, and that we measure the throughput of a system as the number of transactions of each type that run in a given amount of time—say, 1 second. Suppose that system A runs T_1 at 99 transactions per second and T_2 at 1 transaction per second. Similarly, let system B run both T_1 and T_2 at 50 transactions per second. Suppose also that a workload has an equal mixture of the two types of transactions.

If we took the average of the two pairs of numbers (i.e., 99 and 1, versus 50 and 50), it might appear that the two systems have equal performance. However, it is *wrong* to take the averages in this fashion—if we ran 50 transactions of each type, system *A* would take about 50.5 seconds to finish, whereas system *B* would finish in just 2 seconds!

The example shows that a simple measure of performance is misleading if there is more than one type of transaction. The right way to average out the numbers is to take the **time to completion** for the workload, rather than the average *throughput* for each transaction type. We can then compute system performance accurately in transactions per second for a specified workload. Thus, system A takes 50.5/100, which is 0.505 seconds per transaction, whereas system B takes 0.02 seconds per transaction, on average. In terms of throughput, system A runs at an average of 1.98 transactions per second, whereas system B runs at 50 transactions per second. Assuming that transactions of all the types are equally likely, the correct way to average out the throughputs on different transaction types is to take the **harmonic mean** of the throughputs. The harmonic mean of n throughputs $t_1, t_2, \ldots, t_n$ is defined as:

$$\frac{n}{\frac{1}{t_1} + \frac{1}{t_2} + \cdots + \frac{1}{t_n}}$$

For our example, the harmonic mean for the throughputs in system A is 1.98. For system B, it is 50. Thus, system B is approximately 25 times faster than system A on a workload consisting of an equal mixture of the two example types of transactions.

25.2.2 Database-Application Classes

Online transaction processing (**OLTP**) and **decision support**, including **online analytical processing** (**OLAP**), are two broad classes of applications handled by database systems. These two classes of tasks have different requirements. High concurrency and clever

Telegram: @uni_k

www.konkur.in

techniques to speed up commit processing are required for supporting a high rate of update transactions. On the other hand, good query-evaluation algorithms and query optimization are required for decision support. The architecture of some database systems has been tuned to transaction processing; that of others, such as the Teradata series of parallel database systems, has been tuned to decision support. Other vendors try to strike a balance between the two tasks.

Applications usually have a mixture of transaction-processing and decision-support requirements. Hence, which database system is best for an application depends on what mix of the two requirements the application has.

Suppose that we have throughput numbers for the two classes of applications separately, and the application at hand has a mix of transactions in the two classes. We must be careful even about taking the harmonic mean of the throughput numbers because of **interference** between the transactions. For example, a long-running decision-support transaction may acquire a number of locks, which may prevent all progress of update transactions. The harmonic mean of throughputs should be used only if the transactions do not interfere with one another.

25.2.3 The TPC Benchmarks

The **Transaction Processing Performance Council** (**TPC**) has defined a series of benchmark standards for database systems.

The TPC benchmarks are defined in great detail. They define the set of relations and the sizes of the tuples. They define the number of tuples in the relations not as a fixed number, but rather as a multiple of the number of claimed transactions per second, to reflect that a larger rate of transaction execution is likely to be correlated with a larger number of accounts. The performance metric is throughput, expressed as **transactions per second** (**TPS**). When its performance is measured, the system must provide a response time within certain bounds, so that a high throughput cannot be obtained at the cost of very long response times. Further, for business applications, cost is of great importance. Hence, the TPC benchmark also measures performance in terms of **price per TPS**. A large system may have a high number of transactions per second, but it may be expensive (i.e., have a high price per TPS). Moreover, a company cannot claim TPC benchmark numbers for its systems *without* an external audit that ensures that the system faithfully follows the definition of the benchmark, including full support for the ACID properties of transactions.

The first in the series was the **TPC-A benchmark**, which was defined in 1989. This benchmark simulates a typical bank application by a single type of transaction that models cash withdrawal and deposit at a bank teller. The transaction updates several relations—such as the bank balance, the teller's balance, and the customer's balance—and adds a record to an audit-trail relation. The benchmark also incorporates communication with terminals, to model the end-to-end performance of the system realistically. The **TPC-B benchmark** was designed to test the core performance of the database system, along with the operating system on which the system runs. It removes the parts

Telegram: @uni_k

www.konkur.in

of the TPC-A benchmark that deal with users, communication, and terminals, to focus on the backend database server. Neither TPC-A nor TPC-B is in use today.

The **TPC-C benchmark** was designed to model a more complex system than the TPC-A benchmark. The TPC-C benchmark concentrates on the main activities in an order-entry environment, such as entering and delivering orders, recording payments, checking status of orders, and monitoring levels of stock. The TPC-C benchmark is still widely used for benchmarking online transaction processing (OLTP) systems.

The more recent **TPC-E benchmark** is also aimed at OLTP systems but is based on a model of a brokerage firm, with customers who interact with the firm and generate transactions. The firm in turn interacts with financial markets to execute transactions.

The **TPC-D benchmark** was designed to test the performance of database systems on decision-support queries. Decision-support systems are becoming increasingly important today. The TPC-A, TPC-B, and TPC-C benchmarks measure performance on transaction-processing workloads and should not be used as a measure of performance on decision-support queries. The D in TPC-D stands for **decision support**. The TPC-D benchmark schema models a sales/distribution application, with parts, suppliers, customers, and orders, along with some auxiliary information. The sizes of the relations are defined as a ratio, and database size is the total size of all the relations, expressed in gigabytes. TPC-D at scale factor 1 represents the TPC-D benchmark on a 1-gigabyte database, while scale factor 10 represents a 10-gigabyte database. The benchmark workload consists of a set of 17 SQL queries modeling common tasks executed on decision-support systems. Some of the queries make use of complex SQL features, such as aggregation and nested queries.

The benchmark's users soon realized that the various TPC-D queries could be significantly speeded up by using materialized views and other redundant information. There are applications, such as periodic reporting tasks, where the queries are known in advance, and materialized views can be selected carefully to speed up the queries. It is necessary, however, to account for the overhead of maintaining materialized views.

The **TPC-H benchmark** (where ″ represents ad hoc) is a refinement of the TPC-D benchmark. The schema is the same, but there are 22 queries, of which 16 are from TPC-D. In addition, there are two updates, a set of inserts, and a set of deletes. TPC-H prohibits materialized views and other redundant information and permits indices only on primary and foreign keys. This benchmark models ad hoc querying where the queries are not known beforehand, so it is not possible to create appropriate materialized views ahead of time. A variant, TPC-R (where R stands for "reporting"), which is no longer in use, allowed the use of materialized views and other redundant information.

The **TPC-DS benchmark** is a follow-up to the TPC-H benchmark and models the decision-support functions of a retail product supplier, including information about customers, orders, and products, and with multiple sales channels such as retail stores and online sales. It has two subparts of the schema, corresponding to ad hoc querying and reporting, similar to TPC-H and TPC-R. There is a query workload, as well as a data maintenance workload.

Telegram: @uni_k

www.konkur.in

TPC-H and TPC-DS measure performance in this way: The **power test** runs the queries and updates one at a time sequentially, and 3600 seconds divided by the geometric mean of the execution times of the queries (in seconds) gives a measure of queries per hour. The **throughput test** runs multiple streams in parallel, with each stream executing all 22 queries. There is also a parallel update stream. Here the total time for the entire run is used to compute the number of queries per hour.

The **composite query per hour metric**, which is the overall metric, is then obtained as the square root of the product of the power and throughput metrics. A **composite price/performance metric** is defined by dividing the system price by the composite metric.

There are several other TPC benchmarks, such as a data integration benchmark (TPC-DI), benchmarks for big data systems based on Hadoop/Spark (TPCx-HS), and for back-end processing of internet-of-things data (TPCx-IoT).

25.3 Other Issues in Application Development

In this section, we discuss two issues in application development: testing of applications and migration of applications.

25.3.1 Testing Applications

Testing of programs involves designing a **test suite**, that is, a collection of test cases. Testing is not a one-time process, since programs evolve continuously, and bugs may appear as an unintended consequence of a change in the program; such a bug is referred to as program **regression**. Thus, after every change to a program, the program must be tested again. It is usually infeasible to have a human perform tests after every change to a program. Instead, expected test outputs are stored with each test case in a test suite. **Regression testing** involves running the program on each test case in a test suite and checking that the program generates the expected test output.

In the context of database applications, a test case consists of two parts: a database state and an input to a specific interface of the application.

SQL queries can have subtle bugs that can be difficult to catch. For example, a query may execute a join when it should have performed an outer join (i.e., $r \bowtie s$, when it should have actually performed $r ⟖ s$). The difference between these two queries would be found only if the test database had an r tuple with no matching s tuple. Thus, it is important to create test databases that can catch commonly occurring errors. Such errors are referred to as **mutations**, since they are usually small changes to a query (or program). A test case that produces different outputs on an intended query and a mutant of the query is said to **kill the mutant**. A test suite should have test cases that kill (most) commonly occurring mutants.

If a test case performs an update on the database, to check that it executed properly one must verify that the contents of the database match the expected contents. Thus,

Telegram: @uni_k

www.konkur.in

the expected output consists not only of data displayed on the user's screen, but also (updates to) the database state.

Since the database state can be rather large, multiple test cases would share a common database state. Testing is complicated by the fact that if a test case performs an update on the database, the results of other test cases run subsequently on the same database may not match the expected results. The other test cases would then be erroneously reported as having failed. To avoid this problem, whenever a test case performs an update, the database state must be restored to its original state after running the test.

Testing can also be used to ensure that an application meets performance requirements. To carry out such **performance testing**, the test database must be of the same size as the real database would be. In some cases, there is already existing data on which performance testing can be carried out. In other cases, a test database of the required size must be generated; there are several tools available for generating such test databases. These tools ensure that the generated data satisfy constraints such as primary- and foreign-key constraints. They may additionally generate data that look meaningful, for example, by populating a name attribute using meaningful names instead of random strings. Some tools also allow data distributions to be specified; for example, a university database may require a distribution with most students in the range of 18 to 25 years and most faculty in the range of 25 to 65 years.

Even if there is an existing database, organizations usually do not want to reveal sensitive data to an external organization that may be carrying out the performance tests. In such a situation, a copy of the real database may be made, and the values in the copy may be modified in such a way that any sensitive data, such as credit-card numbers, social security numbers, or dates of birth, are **obfuscated**. Obfuscation is done in most cases by replacing a real value with a randomly generated value (taking care to also update all references to that value, in case the value is a primary key). On the other hand, if the application execution depends on the value, such as the date of birth in an application that performs different actions based on the date of birth, obfuscation may make small random changes in the value instead of replacing it completely.

25.3.2 Application Migration

Legacy systems are older-generation application systems that are still in use despite being obsolete. They continue in use due to the cost and risk in replacing them. For example, many organizations developed applications in-house, but they may decide to replace them with a commercial product. In some cases, a legacy system may use old technology that is incompatible with current-generation standards and systems. Some legacy systems in operation today are several decades old and are based on technologies such as databases that use the network or hierarchical data models, or use Cobol and file systems without a database. Such systems may still contain valuable data and may support critical applications.

Replacing legacy applications with new applications is often costly in terms of both time and money, since they are often very large, consisting of millions of lines of code

Telegram: @uni_k

www.konkur.in

developed by teams of programmers, often over several decades. They contain large amounts of data that must be ported to the new application, which may use a completely different schema. Switchover from an old to a new application involves retraining large numbers of staff. Switchover must usually be done without any disruption, with data entered in the old system available through the new system as well.

Many organizations attempt to avoid replacing legacy systems and instead try to interoperate them with newer systems. One approach used to interoperate between relational databases and legacy databases is to build a layer, called a **wrapper**, on top of the legacy systems that can make the legacy system appear to be a relational database. The wrapper may provide support for ODBC or other interconnection standards such as OLE-DB, which can be used to query and update the legacy system. The wrapper is responsible for converting relational queries and updates into queries and updates on the legacy system.

When an organization decides to replace a legacy system with a new system, it may follow a process called **reverse engineering**, which consists of going over the code of the legacy system to come up with schema designs in the required data model (such as an E-R model or an object-oriented data model). Reverse engineering also examines the code to find out what procedures and processes were implemented, in order to get a high-level model of the system. Reverse engineering is needed because legacy systems usually do not have high-level documentation of their schema and overall system design. When coming up with a new system, developers review the design so that it can be improved rather than just reimplemented as is. Extensive coding is required to support all the functionality (such as user interface and reporting systems) that was provided by the legacy system. The overall process is called **re-engineering**.

When a new system has been built and tested, the system must be populated with data from the legacy system, and all further activities must be carried out on the new system. However, abruptly transitioning to a new system, which is called the **big-bang approach**, carries several risks. First, users may not be familiar with the interfaces of the new system. Second, there may be bugs or performance problems in the new system that were not discovered when it was tested. Such problems may lead to great losses for companies, since their ability to carry out critical transactions such as sales and purchases may be severely affected. In some extreme cases the new system has even been abandoned, and the legacy system reused, after an attempted switchover failed.

An alternative approach, called the **chicken-little approach**, incrementally replaces the functionality of the legacy system. For example, the new user interfaces may be used with the old system in the back end, or vice versa. Another option is to use the new system only for some functionality that can be decoupled from the legacy system. In either case, the legacy and new systems coexist for some time. There is therefore a need for developing and using wrappers on the legacy system to provide required functionality to interoperate with the new system. This approach therefore has a higher development cost.

Telegram: @uni_k

www.konkur.in

25.4 Standardization

Standards define the interface of a software system. For example, standards define the syntax and semantics of a programming language, or the functions in an application-program interface, or even a data model (such as the object-oriented database standards). Today, database systems are complex, and they are often made up of multiple independently created parts that need to interact. For example, client programs may be created independently of backend systems, but the two must be able to interact with each other. A company that has multiple heterogeneous database systems may need to exchange data between the databases. Given such a scenario, standards play an important role.

Formal standards are those developed by a standards organization or by industry groups through a public process. Dominant products sometimes become **de facto standards**, in that they become generally accepted as standards without any formal process of recognition. Some formal standards, like many aspects of the SQL-92 and SQL:1999 standards, are **anticipatory standards** that lead the marketplace; they define features that vendors then implement in products. In other cases, the standards, or parts of the standards, are **reactionary standards**, in that they attempt to standardize features that some vendors have already implemented, and that may even have become de facto standards. SQL-89 was in many ways reactionary, since it standardized features, such as integrity checking, that were already present in the IBM SAA SQL standard and in other databases.

Formal standards committees are typically composed of representatives of the vendors and of members from user groups and standards organizations such as the International Organization for Standardization (ISO) or the American National Standards Institute (ANSI), or professional bodies, such as the Institute of Electrical and Electronics Engineers (IEEE). Formal standards committees meet periodically, and members present proposals for features to be added to or modified in the standard. After a (usually extended) period of discussion, modifications to the proposal, and public review, members vote on whether to accept or reject a feature. Some time after a standard has been defined and implemented, its shortcomings become clear and new requirements become apparent. The process of updating the standard then begins, and a new version of the standard is usually released after a few years. This cycle usually repeats every few years, until eventually (perhaps many years later) the standard becomes technologically irrelevant or loses its user base.

This section gives a very high-level overview of different standards, concentrating on the goals of the standard. Detailed descriptions of the standards mentioned in this section appear in the bibliographic notes for this chapter, available online.

25.4.1 SQL Standards

Since SQL is the most widely used query language, much work has been done on standardizing it. ANSI and ISO, with the various database vendors, have played a leading

Telegram: @uni_k

www.konkur.in

role in this work. The SQL-86 standard was the initial version. The IBM Systems Application Architecture (SAA) standard for SQL was released in 1987. As people identified the need for more features, updated versions of the formal SQL standard were developed, called SQL-89 and SQL-92.

The SQL:1999 version of the SQL standard added a variety of features to SQL. We have seen many of these features in earlier chapters.

Subsequent versions of the SQL standard include the following:

- SQL:2003, which is a minor extension of the SQL:1999 standard. Some features such as the SQL:1999 OLAP features (Section 11.3.3) were specified as an amendment to the earlier version of the SQL:1999 standard, instead of waiting for the release of SQL:2003.
- SQL:2006, which added several features related to XML.
- SQL:2008, which introduced only minor extensions to the SQL language such as extensions to the merge clause.
- SQL:2011, which added a number of temporal extensions to SQL, including the ability to associate time periods with tuples, optionally using existing columns as start and end times, and primary key definitions based on the time periods. The extensions support deletes and updates with associated periods; such deletes and updates may result in modification of the time period of existing tuples, along with deletes or inserts of new tuples. A number of operators related to time periods, such as overlaps and contains, were also introduced in SQL:2011.
  In addition, the standard provided a number of other features, such as further extensions to the merge construct, extensions to the window constructs that were introduced in earlier versions of SQL, and extensions to limit the number of results fetched by a query, using a fetch clause.
- SQL:2016, which added a number of features related to JSON support, and support for the aggregate operation listagg, which concatenates attributes from a group of tuples into a string.

It is worth mentioning that most of the new features are supported by only a few database systems, and conversely most database systems support a number of features that are not part of the standard.

25.4.2 Database Connectivity Standards

The **ODBC** standard is a widely used standard for communication between client applications and database systems and defines APIs in several languages. The JDBC standard for communication between Java applications and databases was modeled on ODBC and provides similar functionality.

ODBC is based on the SQL **Call Level Interface** (**CLI**) standards developed by the **X/Open** industry consortium and the SQL Access Group, but it has several extensions.

Telegram: @uni_k

www.konkur.in

The ODBC API defines a CLI, an SQL syntax definition, and rules about permissible sequences of CLI calls. The standard also defines conformance levels for the CLI and the SQL syntax. For example, the core level of the CLI has commands to connect to a database, to prepare and execute SQL statements, to get back results or status values, and to manage transactions. The next level of conformance (level 1) requires support for catalog information retrieval and some other features over and above the core-level CLI; level 2 requires further features, such as the ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

ODBC allows a client to connect simultaneously to multiple data sources and to switch among them, but transactions on each are independent; ODBC does not support two-phase commit.

A distributed system provides a more general environment than a client–server system. The X/Open consortium has also developed the **X/Open XA standards** for interoperation of databases. These standards define transaction-management primitives (such as transaction begin, commit, abort, and prepare-to-commit) that compliant databases should provide; a transaction manager can invoke these primitives to implement distributed transactions by two-phase commit. The XA standards are independent of the data model and of the specific interfaces between clients and databases to exchange data. Thus, we can use the XA protocols to implement a distributed transaction system in which a single transaction can access relational as well as object-oriented databases, yet the transaction manager ensures global consistency via two-phase commit.

There are many data sources that are not relational databases, and in fact may not be databases at all. Examples are flat files and email stores. Microsoft's **OLE-DB** is a C++ API with goals similar to ODBC, but for nondatabase data sources that may provide only limited querying and update facilities. Just like ODBC, OLE-DB provides constructs for connecting to a data source, starting a session, executing commands, and getting back results in the form of a rowset, which is a set of result rows.

The **ActiveX Data Objects** (**ADO**) and **ADO.NET** APIs, created by Microsoft, provide an interface to access data from not only relational databases, but also some other types of data sources, such as OLE-DB data sources.

25.4.3 Object Database Standards

Standards in the area of object-oriented databases (OODB) have so far been driven primarily by OODB vendors. The **Object Database Management Group** (**ODMG**) was a group formed by OODB vendors to standardize the data model and language interfaces to OODBs. ODMG is no longer active. JDO is a standard for adding persistence to Java.

There were several other attempts to standardize object databases and related object-based technologies such as services. However, most were not widely adopted, and they are rarely used anymore.

Object-relational mapping technologies, which store data in relational databases at the back end but provide programmers with an object-based to access and manip-

Telegram: @uni_k

www.konkur.in

ulate data, have proven quite popular. Systems that support object-relational mapping include Hibernate, which supports Java, and the data layer of the popular Django Web framework, which is based on the Python programming language. However, there are no widely accepted formal standards in this area.

25.5 Distributed Directory Systems

Consider an organization that wishes to make data about its employees available to a variety of people in the organization; examples of the kinds of data include name, designation, employee-id, address, email address, phone number, fax number, and so on. Such data are often shared via directories, which allow users to browse and search for desired information.

In general, a directory is a listing of information about some class of objects such as persons. Directories can be used to find information about a specific object, or in the reverse direction to find objects that meet a certain requirement.

A major application of directories today is to authenticate users: applications can collect authentication information such as passwords from users and authenticate them using the directory. Details about the user category (e.g., is the user a student or an instructor), as well as authorizations that a user has been given, may also be shared through a directory. Multiple applications in an organization can then authenticate users using a common directory service and user category and authorization information from the directory to provide users only with data that they are authorized to see.

Directories can be used for storing other types of information, much like file system directories. For instance, web browsers can store personal bookmarks and other browser settings in a directory system. A user can thus access the same settings from multiple locations, such as at home and at work, without having to share a file system.

25.5.1 Directory Access Protocols

Directory information can be made available through web interfaces, as many organizations, and phone companies in particular, do. Such interfaces are good for humans. However, programs too need to access directory information.

Several **directory access protocols** have been developed to provide a standardized way of accessing data in a directory. The most widely used among them today is the **Lightweight Directory Access Protocol** (**LDAP**).

All the types of data in our examples can be stored without much trouble in a database system and accessed through protocols such as JDBC or ODBC. The question then is, why come up with a specialized protocol for accessing directory information? There are at least two answers to the question.

- First, directory access protocols are simplified protocols that cater to a limited type of access to data. They evolved in parallel with the database access protocols.

Telegram: @uni_k

www.konkur.in

- Second, and more important, directory systems provide a simple mechanism to name objects in a hierarchical fashion, similar to file system directory names, which can be used in a distributed directory system to specify what information is stored in each of the directory servers. For example, a particular directory server may store information for Bell Laboratories employees in Murray Hill, while another may store information for Bell Laboratories employees in Bangalore, giving both sites autonomy in controlling their local data. The directory access protocol can be used to obtain data from both directories across a network. More important, the directory system can be set up to automatically forward queries made at one site to the other site, without user intervention.

For these reasons, several organizations have directory systems to make organizational information available online through a directory access protocol. Information in an organizational directory can be used for a variety of purposes, such as to find addresses, phone numbers, or email addresses of people, to find which departments people are in, and to track department hierarchies.

As may be expected, several directory implementations find it beneficial to use relational databases to store data instead of creating special-purpose storage systems.

25.5.2 LDAP: Lightweight Directory Access Protocol

In general a directory system is implemented as one or more servers, which service multiple clients. Clients use the API defined by the directory system to communicate with the directory servers. Directory access protocols also define a data model and access control. The **X.500 directory access protocol**, defined by the International Organization for Standardization (ISO), is a standard for accessing directory information. However, the protocol is rather complex and is not widely used. The **Lightweight Directory Access Protocol** (**LDAP**) provides many of the X.500 features, but with less complexity, and is widely used. In addition to several open-source implementations, the Microsoft Active Directory system, which is based on LDAP, is used in a large number of organizations.

In the rest of this section, we shall outline the data model and access protocol details of LDAP.

25.5.2.1 LDAP Data Model

In LDAP, directories store **entries**, which are similar to objects. Each entry must have a **distinguished name** (**DN**), which uniquely identifies the entry. A DN is in turn made up of a sequence of **relative distinguished names** (**RDNs**). For example, an entry may have the following distinguished name:

cn=Silberschatz, ou=Computer Science, o=Yale University, c=USA

As you can see, the distinguished name in this example is a combination of a name and (organizational) address, starting with a person's name, then giving the organizational unit (ou), the organization (o), and country (c). The order of the components of a distinguished name reflects the normal postal address order, rather than the reverse

Telegram: @uni_k

www.konkur.in

order used in specifying path names for files. The set of RDNs for a DN is defined by the schema of the directory system.

Entries can also have attributes. LDAP provides binary, string, and time types, and additionally the types tel for telephone numbers, and PostalAddress for addresses (lines separated by a "$" character). Unlike those in the relational model, attributes are multivalued by default, so it is possible to store multiple telephone numbers or addresses for an entry.

LDAP allows the definition of **object classes** with attribute names and types. Inheritance can be used in defining object classes. Moreover, entries can be specified to be of one or more object classes. It is not necessary that there be a single most-specific object class to which an entry belongs.

Entries are organized into a **directory information tree** (**DIT**), according to their distinguished names. Entries at the leaf level of the tree usually represent specific objects. Entries that are internal nodes represent objects such as organizational units, organizations, or countries. The children of a node have a DN containing all the RDNs of the parent, and one or more additional RDNs. For instance, an internal node may have a DN c=USA, and all entries below it have the value USA for the RDN c.

The entire distinguished name need not be stored in an entry. The system can generate the distinguished name of an entry by traversing up the DIT from the entry, collecting the RDN=value components to create the full distinguished name.

Entries may have more than one distinguished name—for example, an entry for a person in more than one organization. To deal with such cases, the leaf level of a DIT can be an **alias** that points to an entry in another branch of the tree.

25.5.2.2 Data Manipulation

Unlike SQL, LDAP does not define either a data-definition language or a data-manipulation language. However, LDAP defines a network protocol for carrying out data definition and manipulation. Users of LDAP can either use an application-programming interface or use tools provided by various vendors to perform data definition and manipulation. LDAP also defines a file format called **LDAP Data Interchange Format** (**LDIF**) that can be used for storing and exchanging information.

The querying mechanism in LDAP is very simple, consisting of just selections and projections, without any join. A query must specify the following:

- A base—that is, a node within a DIT—by giving its distinguished name (the path from the root to the node).
- A search condition, which can be a Boolean combination of conditions on individual attributes. Equality, matching by wild-card characters, and approximate equality (the exact definition of approximate equality is system dependent) are supported.
- A scope, which can be just the base, the base and its children, or the entire subtree beneath the base.

Telegram: @uni_k

www.konkur.in

- Attributes to return.
- Limits on number of results and resource consumption.

The query can also specify whether to automatically dereference aliases; if alias dereferences are turned off, alias entries can be returned as answers.

We omit further details of query support in LDAP but note that LDAP implementations support an API for querying/updating LDAP data and may additionally support web services for querying LDAP data.

25.5.2.3 Distributed Directory Trees

Information about an organization may be split into multiple DITs, each of which stores information about some entries. The **suffix** of a DIT is a sequence of RDN=value pairs that identify what information the DIT stores; the pairs are concatenated to the rest of the distinguished name generated by traversing from the entry to the root. For instance, the suffix of a DIT may be o=Nokia, c=USA, while another may have the suffix o=Nokia, c=India. The DITs may be organizationally and geographically separated.

A node in a DIT may contain a **referral** to another node in another DIT; for instance, the organizational unit Bell Labs under o=Nokia, c=USA may have its own DIT, in which case the DIT for o=Nokia, c=USA would have a node ou=Bell Labs representing a referral to the DIT for Bell Labs.

Referrals are the key component that help organize a distributed collection of directories into an integrated system. When a server gets a query on a DIT, it may return a referral to the client, which then issues a query on the referenced DIT. Access to the referenced DIT is transparent, proceeding without the user's knowledge. Alternatively, the server itself may issue the query to the referred DIT and return the results along with locally computed results.

The hierarchical naming mechanism used by LDAP helps break up control of information across parts of an organization. The referral facility then helps integrate all the directories in an organization into a single virtual directory.

Although it is not an LDAP requirement, organizations often choose to break up information either by geography (for instance, an organization may maintain a directory for each site where the organization has a large presence) or by organizational structure (for instance, each organizational unit, such as department, maintains its own directory). Many LDAP implementations support master–slave and multimaster replication of DITs.

25.6 Summary

- Tuning of the database-system parameters, as well as the higher-level database design—such as the schema, indices, and transactions—is important for good performance. Tuning is best done by identifying bottlenecks and eliminating them.

Telegram: @uni_k

www.konkur.in

- Database tuning can be done at the level of schema and queries, at the level of database system parameters, and at the level of hardware. Database systems usually have a variety of tunable parameters, such as buffer sizes.
- The right choice of indices and materialized views, and the use of horizontal partitioning can provide significant performance benefits. Tools for automated tuning based on workload history can help significantly in such tuning. The set of indices and materialized views can be appropriately chosen to minimize overall cost. Vertical partitioning, and columnar storage can lead to significant benefits in online analytical processing systems.
- Transactions can be tuned to minimize lock contention; snapshot isolation and sequence numbering facilities supporting early lock release are useful tools for reducing read-write and write-write contention.
- Hardware tuning includes choice of memory size, the use of SSDs versus magnetic hard disks, and increasingly, the number of CPU cores.
- Performance benchmarks play an important role in comparisons of database systems, especially as systems become more standards compliant. The TPC benchmark suites are widely used, and the different TPC benchmarks are useful for comparing the performance of databases under different workloads.
- Applications need to be tested extensively as they are developed and before they are deployed. Testing is used to catch errors as well as to ensure that performance goals are met.
- Legacy systems are systems based on older-generation technologies such as nonrelational databases or even directly on file systems. Interfacing legacy systems with new-generation systems is often important when they run mission-critical systems. Migrating from legacy systems to new-generation systems must be done carefully to avoid disruptions, which can be very expensive.
- Standards are important because of the complexity of database systems and their need for interoperation. Formal standards exist for SQL. De facto standards, such as ODBC and JDBC, and standards adopted by industry groups have played an important role in the growth of client–server database systems.
- Distributed directory systems have played an important role in many applications, and can be viewed as distributed databases. LDAP is widely used for authentication and for tracking employee information in organizations.

Review Terms

- Performance tuning
- Bottlenecks
- Queueing systems
- Tuning of physical schema

Telegram: @uni_k

www.konkur.in

- Tuning of indices
- Materialized views
- Immediate view maintenance
- Deferred view maintenance
- Tuning of physical design
- Workload
- Tuning of queries
- Set orientation
- Batch update (JDBC)
- Bulk load
- Bulk update
- Merge statement
- Tuning of logical schema
- Tunable parameters
- Tuning of concurrent transactions
- Sequences
- Minibatch transactions
- Tuning of hardware
- Five minute rule
- Performance simulation
- Performance benchmarks
- Service time
- Throughput
- Database-application classes
- OLTP
- Decision support
- The TPC benchmarks
  - TPC-C
  - TPC-D
  - TPC-E
  - TPC-H
  - TPC-DS
- Regression testing
- Killing mutants
- Application migration
- Legacy systems
- Reverse engineering
- Re-engineering
- Standardization
  - Formal standards
  - De facto standards
  - Anticipatory standards
  - Reactionary standards
- Database connectivity standards
- X/Open XA standards
- Object database standards
- XML-based standards
- LDAP
- Directory information tree
- Distributed directory trees

Practice Exercises

25.1 Find out all performance information your favorite database system provides. Look for at least the following: what queries are currently executing or executed recently, what resources each of them consumed (CPU and I/O), what fraction of page requests resulted in buffer misses (for each query, if available), and what locks have a high degree of contention. Also get information about CPU, I/O and network utilization, including the number of open network connections using your operating system utilities.

Telegram: @uni_k

www.konkur.in

25.2 Many applications need to generate sequence numbers for each transaction.

a. If a sequence counter is locked in two-phase manner, it can become a concurrency bottleneck. Explain why this may be the case.

b. Many database systems support built-in sequence counters that are not locked in two-phase manner; when a transaction requests a sequence number, the counter is locked, incremented and unlocked.

i. Explain how such counters can improve concurrency.

ii. Explain why there may be gaps in the sequence numbers belonging to the final set of committed transactions.

25.3 Suppose you are given a relation $r(a, b, c)$.

a. Give an example of a situation under which the performance of equality selection queries on attribute a can be greatly affected by how r is clustered.

b. Suppose you also had range selection queries on attribute b. Can you cluster r in such a way that the equality selection queries on $r.a$ and the range selection queries on $r.b$ can both be answered efficiently? Explain your answer.

c. If clustering as above is not possible, suggest how both types of queries can be executed efficiently by choosing appropriate indices.

25.4 When a large number of records are inserted into a relation in a short period of time, it is often recommended that all indices be dropped, and recreated after the inserts have been completed.

a. What is the motivation for this recommendation?

b. Dropping and recreation of indices can be avoided by bulk-updating of the indices. Suggest how this could be done efficiently for B^+-tree indices.

c. If the indices were write-optimized indices such as LSM trees, would this advice be meaningful?

25.5 Suppose that a database application does not appear to have a single bottleneck; that is, CPU and disk utilization are both high, and all database queues are roughly balanced. Does that mean the application cannot be tuned further? Explain your answer.

25.6 Suppose a system runs three types of transactions. Transactions of type A run at the rate of 50 per second, transactions of type B run at 100 per second, and transactions of type C run at 200 per second. Suppose the mix of transactions has 25 percent of type A, 25 percent of type B, and 50 percent of type C.

Telegram: @uni_k

www.konkur.in

a. What is the average transaction throughput of the system, assuming there is no interference between the transactions?

b. What factors may result in interference between the transactions of different types, leading to the calculated throughput being incorrect?

25.7 Suppose an application programmer was supposed to write a query

```
select *
from r natural left outer join s;
```

on relations $r(A, B)$ and $s(B, C)$, but instead wrote the query

```
select *
from r natural join s;
```

a. Give sample data for r and s on which both queries would give the same result.

b. Give sample data for r and s where the two queries would give different results, thereby exposing the error in the query,

25.8 List some benefits and drawbacks of an anticipatory standard compared to a reactionary standard.

25.9 Describe how LDAP can be used to provide multiple hierarchical views of data, without replicating the base-level data.

Exercises

25.10 Database tunning:

a. What are the three broad levels at which a database system can be tuned to improve performance?

b. Give two examples of how tuning can be done for each of the levels.

25.11 When carrying out performance tuning, should you try to tune your hardware (by adding disks or memory) first, or should you try to tune your transactions (by adding indices or materialized views) first. Explain your answer.

25.12 Suppose that your application has transactions that each access and update a single tuple in a very large relation stored in a B^+-tree file organization. Assume that all internal nodes of the B^+-tree are in memory, but only a very small fraction of the leaf pages can fit in memory. Explain how to calculate the minimum number of disks required to support a workload of 1000 transactions

Telegram: @uni_k

www.konkur.in

per second. Also calculate the required number of disks, using values for disk parameters given in Section 12.3.

25.13 What is the motivation for splitting a long transaction into a series of small ones? What problems could arise as a result, and how can these problems be averted?

25.14 Suppose the price of memory falls by half, and the speed of disk access (number of accesses per second) doubles, while all other factors remain the same. What would be the effect of this change on the 5-minute and 1-minute rule?

25.15 List at least four features of the TPC benchmarks that help make them realistic and dependable measures.

25.16 Why was the TPC-D benchmark replaced by the TPC-H and TPC-R benchmarks?

25.17 Explain what application characteristics would help you decide which of TPC-C, TPC-H, or TPC-R best models the application.

25.18 Given that the LDAP functionality can be implemented on top of a database system, what is the need for the LDAP standard?

Further Reading

[Harchol-Balte (2013)] provides textbook coverage of queuing theory from a computer science perspective.

Information about tuning support in IBM DB2, Oracle and Microsoft SQL Server may be found in their respective manuals online, as well as in numerous books. [Shasha and Bonnet (2002)] provides detailed coverage of database tuning principles. [O'Neil and O'Neil (2000)] provides a very good textbook coverage of performance measurement and tuning. The 5-minute and 1-minute rules are described in [Gray and Graefe (1997)], [Graefe (2008)], and [Appuswamy et al. (2017)].

An early proposal for a database-system benchmark (the Wisconsin benchmark) was made by [Bitton et al. (1983)]. The TPC-A, TPC-B, and TPC-C benchmarks are described in [Gray (1991)]. An online version of all the TPC benchmark descriptions, as well as benchmark results, is available on the World Wide Web at the URL www.tpc.org; the site also contains up-to-date information about new benchmark proposals.

The XData system (www.cse.iitb.ac.in/infolab/xdata) provides tools for generating test data to catch errors in SQL queries, as well as for grading student SQL queries.

A number of standards documents, including several parts of the SQL standard, can be found on the ISO/IEC website (standards.iso.org/ittf/PubliclyAvailableStandards/index.html). Information about ODBC, OLE-DB, ADO, and ADO.NET can be found on the web site

Telegram: @uni_k

www.konkur.in

www.microsoft.com/data. A wealth of information on XML-based standards and tools is available online on the web site www.w3c.org.

Bibliography

[Appuswamy et al. (2017)] R. Appuswamy, R. Borovica, G. Graefe, and A. Ailamaki, "The Five minute Rule Thirty Years Later and its Impact on the Storage Hierarchy", In *Proceedings of the 7th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures* (2017).

[Bitton et al. (1983)] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", In *Proc. of the International Conf. on Very Large Databases* (1983), pages 8–19.

[Graefe (2008)] G. Graefe, "The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules", *ACM Queue*, Volume 6, Number 4 (2008), pages 40–52.

[Gray (1991)] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd edition, Morgan Kaufmann (1991).

[Gray and Graefe (1997)] J. Gray and G. Graefe, "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb", *SIGMOD Record*, Volume 26, Number 4 (1997), pages 63–68.

[Harchol-Balte (2013)] M. Harchol-Balte, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, Cambridge University Press (2013).

[O'Neil and O'Neil (2000)] P. O'Neil and E. O'Neil, *Database: Principles, Programming, Performance*, 2nd edition, Morgan Kaufmann (2000).

[Shasha and Bonnet (2002)] D. Shasha and P. Bonnet, *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann (2002).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

CHAPTER 26



Blockchain Databases

At the most basic level, a blockchain provides an alternative data format for storing a database, and its paradigm for transaction processing enables a high level of decentralization.

A major application of blockchain technology is in the creation of **digital ledgers**. A ledger in the financial world is a book of financial accounts, that keeps track of transactions. For example, each time you deposit or withdraw money from your account, an entry is added to a ledger maintained by the bank. Since the ledger is maintained by the bank, a customer of the bank implicitly trusts the bank to not cheat by adding unauthorized transactions to the ledger, such as an unauthorized withdrawal, or modifying the ledger by deleting transactions such as a deposit.

Blockchain-based distributed ledgers maintain a ledger cooperatively among several parties, in such a way that each transaction is digitally signed as proof of authenticity, and further, the ledger is maintained in such a way that once entries are added, they cannot be deleted or modified by one party, without detection by others.

Blockchains form a key foundation of Bitcoin and other cryptocurrencies. Although much of the technology underlying blockchains was initially developed in the 1980s and 1990s, blockchain technology gained widespread popular attention in the 2010s as a result of boom (and subsequent bust) in Bitcoin and other cryptocurrencies.

However, beyond the many cryptocurrency schemes, blockchains can provide a secure data-storage and data-processing foundation for business applications, without requiring complete trust in any one party. For example, consider a large corporation and its suppliers, all of whom maintain data about where products and components are located at any time as part of the manufacturing process. Even if the organizations are presumed trustworthy, there may a situation where one of them has a strong incentive to cheat and rewrite the record. A blockchain can help protect from such fraudulent updates. Ownership documents, such as real-estate deeds, are another example of the potential for blockchain use. Criminals may commit real-estate fraud by creating fake ownership deeds, which could allow them to sell a property that they do not own, or could allow the same property to be sold multiple times by an actual owner. Blockchains can help verify the authenticity of digitally represented ownership documents; blockchains can also ensure that once an owner has sold a property, the

Telegram: @uni_k

www.konkur.in

owner cannot sell it again to another person without getting detected. The security provided by the blockchain data structure makes it possible to allow the public to view these real-estate records without putting them at risk. We describe other applications for blockchains later in the chapter.

In this chapter, we shall look at blockchain from a database perspective. We shall identify the ways in which blockchain databases differ from the traditional databases we have studied elsewhere in this book and show how these distinguishing features are implemented. We shall consider alternatives to Bitcoin-style algorithms and implementation that are more suited to an enterprise database environment. With this database-oriented focus, we shall not consider the financial implications of cryptocurrencies, nor the issues of managing one's holding of such currencies via a cryptocurrency wallet or exchange.

26.1 Overview

Before we study blockchains in detail, we first give an overview of cryptocurrencies, which have driven the development and usage of blockchains. We note, however, that blockchains have many uses beyond cryptocurrencies.

Traditional currencies, also known as "fiat currencies" are typically issued by a central bank of a country, and guaranteed by the government of that country. Currency notes are at one level just a piece of paper; the only reason they are of value is that the government that issues the currency guarantees the value of the currency, and users trust the government. Today, although financial holdings continue to be denominated in terms of a currency, most of the financial holdings are not physically present as currency notes; they are merely entries in the ledger of a bank or other financial institution. Users of the currency are forced to trust the organization that maintains the ledger.

A **cryptocurrency** is a currency created purely online, and recorded in a way that does not require any one organization (or country) to be totally trusted. This term arises from the fact that any such scheme has to based on encryption technologies. Since any digital information can be copied easily, unlike currency notes, any cryptocurrency scheme must be able to prevent "double spending" of money. To solve this problem, cryptocurrencies use ledgers to record transactions. Further, the ledgers are stored a secure, distributed infrastructure, with no requirement to trust any one party. These two key concepts, decentralization and trustlessness, are fundamental to cryptocurrencies. Cryptocurrenies typically aim, like regular currency, and unlike credit card or debit card transactions, to provide transaction anonymity, to preserve the privacy of users of the currency. However, since cryptocurrency blockchains are public data analytics may be used to compromise or limit anonymity.

Bitcoin, which was the first successful cryptocurrency, emerged with the publication of a paper by Satoshi Nakamoto[1] in 2008 and the subsequent publication of the open-source Bitcoin code in 2009. The ideas in the original bitcoin paper solved

[1]Satoshi Nakamoto is a pseudonym for a person or group that anonymously

Telegram: @uni_k

www.konkur.in

a number of problems, and thereby allowed cryptocurrencies, which had earlier been considered impractical, to become a reality.

However, the underlying concepts and algorithms in many cases go back decades in their development. The brilliance of Nakamoto's work was a combination of innovation and well-architected use of prior research. The successes of Bitcoin prove the value of this contribution, but the target—an anonymous, trustless, fully distributed concurrency—drove many technical decisions in directions that work less well in a database setting. The Further Reading section at the end of the chapter cites key historical papers in the development of these ideas.

At its most basic level, a blockchain is a linked list of blocks of data that can be thought of as constituting a log of updates to data. What makes blockchain technology interesting is that blockchains can be managed in a distributed manner in such a way that they are highly tamper resistant, and cannot be easily modified or manipulated by any one participant, except by appending digitally signed records to the blockchain.

In a business setting, trustless distributed control is valuable, but absolute anonymity runs counter to both principles of accounting and regulatory requirements. This leads to two main scenarios for the use of blockchains. Bitcoin's blockchain is referred to as a **public blockchain**, since it allows any site to join and participate in the tasks of maintaining the blockchain. In contrast, most enterprise blockchains are more restricted and referred to as **permissioned blockchains**. In a permissioned blockchain, participation is not open to the public. Access is granted by a permissioning authority, which may be an enterprise, a consortium of enterprises, or a government agency.

Bitcoin introduced a number of ideas that made public blockchains practical, but these have a significant cost in terms of CPU power (and thereby, electrical power) needed to run the blockchain, as well as latencies in processing transactions. By relaxing Bitcoin's strong assumptions about trustlessness and anonymity, it is possible to overcome many of the inefficiencies and high latencies of the Bitcoin model and design blockchains that further the goals of enterprise data management.

In this chapter, we begin by looking at the classic blockchain structure as used in Bitcoin and use that to introduce the key distinguishing properties of a blockchain. Achieving many of these properties relies upon one-way cryptographic hash functions. These hash functions are quite different from those used in Chapter 24 as a means of indexing databases. Cryptographic hash functions need to have some specific mathematical properties such as the following: given a data value x and a hash function h, it must be relatively easy to compute $h(x)$ but virtually impossible to find x given $h(x)$.

When a blockchain is stored distributed across multiple systems, an important issue is to ensure that the participating systems agree on what are the contents of the blockchain, and what gets added to it at each step. When participants trust each other, but may be vulnerable to failure, consensus techniques that we studied earlier in Section 23.8 can be used to ensure that all participants agree on the contents of a log (and a blockchain is, at its core, a log). However, reaching agreement on what data get added to a blockchain is much more challenging when participants in the blockchain do not trust each other and have no centralized control. The basic consensus algorithms are

Telegram: @uni_k

www.konkur.in

not applicable in such a setting. For example, an attacker could create a large number of systems, each of which joins the blockchain as a participant; the attacker could thereby control a majority of the participating systems. Any decision based on a majority can then be controlled by the attacker, who can force decisions that can tamper with the contents of the blockchain. The tamper resistance property of the blockchain would then be compromised.

We begin by describing the energy-intensive approach of Bitcoin, but we then consider a variety of alternative, more efficient approaches used in other cryptocurrencies. Finally, we consider approaches based on Byzantine-consensus algorithms, which are consensus algorithms that are resistant to some fraction of the participating nodes not just failing, but also lying and attempting to disrupt consensus. Byzantine consensus is well-suited to an enterprise blockchain environment, and can be used if the blockchain is *permissioned*, that is, some organization controls who can have permission to access the blockchain. Byzantine consensus is an old problem and solutions have been around for many years. However, the special constraints of blockchain databases have led to some newer approaches to this problem. References to more details on Byzantine consensus techniques may be found in the Further Reading section at the end of the chapter.

Blockchain databases store more than just currency-based debit-credit transactions. Like any database, they may store a variety of types of data about the enterprise. A traditional blockchain data organization makes it difficult to retrieve such data efficiently, but pairing a blockchain with a traditional database or building the blockchain on top of a database can enable faster query processing. We shall explore a variety of means of speeding up not only queries but also update transactions, both within the blockchain itself and by performing certain operations "off chain" and adding them in bulk to the blockchain at a later time.

After covering blockchain algorithms, we shall explore (in Section 26.8) some of the most promising applications of blockchain databases.

26.2 Blockchain Properties

At its most basic level, a blockchain is a linked list of blocks of data. A distinguishing feature of the blockchain data structure is that the pointers in the linked list include not only the identifier of the next older block, but also a hash of that older block. This structure is shown in Figure 26.1. The initial block, or **genesis block**, is shown as block 0 in the figure. It is set up by the creator of the blockchain. Each time a block is added to the chain, it includes the pair of values (pointer-to-previous-block, hash-of-previous-block). As a result, any change made to block is easily detected by comparing a hash of that block to the hash value contained in the next block in the chain. The hash value in the next block could be changed, but then the block after that would also have to be changed, and so on.

Telegram: @uni_k

www.konkur.in

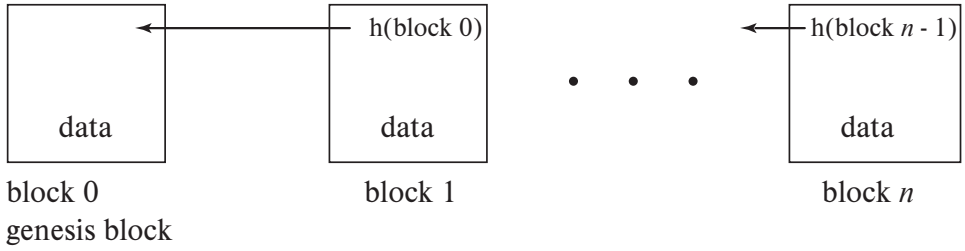


Figure 26.1 Blockchain data structure.

This hash-validated pointer format in a blockchain makes tampering with a blockchain hard. To make tampering virtually impossible, it is necessary to ensure that any tampering with the blockchain is easily detected and that the correct version of the blockchain is easily determined. To achieve this, the hash function must have certain mathematical properties that we shall discuss shortly. Further, the chain itself must be replicated and distributed among many independent nodes so that no single node or small group of nodes can tamper with the blockchain. Since the blockchain is replicated across multiple nodes, a distributed consensus algorithm needs to be used to maintain agreement regarding the correct current state of the blockchain. In this way, even if some nodes try to tamper with the blockchain contents, as long as a majority are honest, making decisions based on a majority vote can ensure the integrity of the blockchain.

The above approach works if the set of nodes that participates in the blockchain is controlled in some fashion that makes it difficult for an adversary to control a majority of the nodes. However, such control goes against the goal of not have any central control, and is viewed as unacceptable in public blockchains such as Bitcoin, which are based on public blockchains in which the number of participating nodes may change continuously. Any computer may download the blockchain and attempt to add blocks (the code for implementing blockchains is available in open source). As a result, a majority-based approach can be overwhelmed by an adversary who sets up a large number of low-cost computers as nodes. Such an attack is called a **Sybil attack**.

The way in which consensus is achieved among independent nodes varies among blockchains. The variations address trade-offs between performance (latency and throughput) and robustness to adversarial attacks on the consensus mechanism, including Sybil attacks. When we addressed distributed consensus in Chapter 23, we assumed that a single organization controlled the entire distributed system, and so the consensus algorithm had to tolerate only possible failures of nodes or the network that were fail-stop, where participants do not behave in an adversarial manner.

In a typical blockchain application, the chain is shared among multiple independent organizations. In the extreme case, for example Bitcoin, anyone can set up a node and participate, possibly for nefarious purposes. This implies that the types of failure that may occur are not just cases where a device or system stops working, but also cases

Telegram: @uni_k

www.konkur.in

where a system remains operational but behaves in an adversarial manner. In most enterprise settings, the blockchain is **permissioned**, providing some control over the set of participants, but still without direct controls to prevent malicious behavior.

A node participating in a blockchain fully needs to participate in the consensus mechanism and maintain its own replica of the blockchain. Such a node is called a **full node**. In some applications, there is a need for low-cost nodes that submit updates to the blockchain, but do not have the storage or computational power to participate in the consensus process. Such a node is called a **light node**.

We discuss blockchain consensus algorithms in detail in Section 26.4. Blockchain consensus algorithms can be placed into one of several broad categories:

- **Proof of work:** Proof of work, which is described in detail in Section 26.4.1, provides a solution to Sybil attacks by making it very expensive for an attacker to control a majority of the nodes. Specifically, the nodes agree that the next block on the blockchain will be added by the first node to solve a certain hard mathematical problem. This is referred to as **mining** a block. Proof-of-work algorithms are robust to adversarial behavior as long as the adversary does not control more than half the computing power in the entire network. To ensure this requirement, the problems are made intentionally hard, and require a lot of computational effort. Thus, robustness comes at the price of a huge amount of otherwise useless computation along with the price of electricity needed to carry out the computation.
- **Proof of stake:** Proof of stake, which is described in Section 26.4.2, provides another solution to Sybil attacks. Here, the nodes agree to select the next node to add a block to the blockchain based on an amount of the blockchain's currency owned or held in reserve by a node.
- **Byzantine consensus:** Byzantine consensus does not solve the problem of Sybil attacks, but can be used in non-public blockchains, where entry of nodes to the system can be controlled. While some nodes may behave maliciously, it is assumed that a substantial majority are honest. In Byzantine consensus, described in Section 26.4.3, the next node to add a block to the blockchain is decided by an algorithm from the class of algorithms referred to as Byzantine-consensus algorithms. Like the basic consensus algorithms we described earlier in Section 23.8, these algorithms achieve agreement by message passing, but unlike those algorithms, these algorithms can tolerate some number of nodes being malicious by either disrupting consensus or trying to cause an incorrect consensus to be reached. This approach requires significantly more messages to be exchanged than in the case of the basic consensus algorithms of Section 26.4.3, but this is a worthwhile trade-off for the ability for a system to work correctly in the presence of a certain number of malicious nodes.
- **Other approaches:** There are several other less widely used consensus mechanisms, some of which are variants of the preceding mechanisms. These include proof of

Telegram: @uni_k

www.konkur.in

activity, proof of burn, proof of capacity, and proof of elapsed time. See the Further Reading section at the end of the chapter for details.

Another way to damage a blockchain besides attempting to alter blocks is to add a new block to a block other than the most recent one. This is called a **fork**. Forking may occur due to malicious activity, but there are two sources of nonmalicious forks:

1. Two distinct nodes may add a new block after the most recent block, but they do it so close together in time that both are added successfully, thus creating a forked chain. These accidental forks are resolved by a protocol rule that nodes always attempt to add blocks to the end of the longest chain. This probabilistically limits these accidental forks to a short length. The blocks on the shorter forks are said to be *orphaned*, and the contents of those blocks will get inserted on the real chain later if those contents are not already there.
2. A majority of blockchain users may agree to fork the blockchain in order to change some aspect of the blockchain protocol or data structure. This is a rare event and one that, when it has occurred in major blockchains, has caused major controversy. Such a fork is said to be a **soft fork** if prior blocks are not invalidated by the fork. That is, the old version of the blockchain software will recognize blocks from the new version as valid. This permits a gradual transition from the old version of the blockchain software to the new version. In a **hard fork**, the old version of the blockchain software will deem blocks from the new version to be invalid. After a hard fork, if the old version of the blockchain software remains in use, it will lead to a separate blockchain with different contents.

Because of the possibility of orphaned blocks, it may be necessary to wait for several additional blocks to be added before it is safe to assume s block will not be orphaned.

Note 26.1 on page 1258 presents a few examples of notable blockchain forks.

So far, we have not said much about the actual data in the blocks. The contents of blocks vary by application domain. In a cryptocurrency application, the most common data found in blocks are basic currency-transfer transactions. Since any node can add a block, there needs to be a way to ensure that transactions entered are in fact genuine. This is achieved via a technique called a **digital signature** that allows a user to "sign" a transaction and allows every node to verify that signature. This prevents fake transactions from being added to the chain and prevents participants in the transaction from subsequently denying their involvement in the transaction. This latter property is referred to as **irrefutability**.

Transactions are broadcast to all nodes participating in the blockchain; when a node adds a block to the chain, the block contains all transactions received by the node that have not already been added to the chain.

The users who submit transactions may be known to the blockchain administrator in a permissioned blockchain, but in a public blockchain like Bitcoin, there is no direct

Telegram: @uni_k

www.konkur.in



connection between a user ID and any real-world entity. This anonymity property is a key feature of Bitcoin, but its value is diminished because of the possibility to tie a user ID to some off-chain activity, thereby de-anonymizing the user. De-anonymization can occur if the user enters into a transaction with a user whose user ID has already been de-anonymized. De-anonymization can occur also via data mining on the blockchain data and correlating on-chain activity by a specific user ID with the "real-world" activity of a specific individual.

Finally, a feature of blockchains is the ability to store executable code, referred to as a **smart contract**. A smart contract can implement complex transactions, take action at some point in the future based on specified conditions, and, more generally, encode a complex agreement among a set of users. Blockchains differ not only in the language used for smart contracts but also in the power of the language used. Many are Turing complete, but some (notably, Bitcoin) have more limited power. We discuss smart contracts, including how and when their code is executed, in Section 26.6.

Telegram: @uni_k

www.konkur.in

We summarize this discussion by listing a set of properties of blockchains:[2]

- **Decentralization:** In a public blockchain, control of the blockchain is by majority consensus with no central controlling authority. In a permissioned blockchain, the degree of central control is limited, typically only to access authorization and identity management. All other actions happen in a decentralized manner.
- **Tamper Resistance:** Without gaining control over a majority of the blockchain network, it is infeasible to change the contents of blocks.
- **Irrefutability:** Activity by a user on a blockchain is signed cryptographically by the user. These signatures can be validated easily by anyone and thus prove that the user indeed is responsible for the transaction.
- **Anonymity:** Users of a blockchain have user IDs that are not tied directly to any personally identifying information, though anonymity may be compromised indirectly. Permissioned blockchains may offer only limited anonymity or none at all.

26.3 Achieving Blockchain Properties via Cryptographic Hash Functions

In this section, we focus on the use of cryptographic hash functions to ensure some of the properties of blockchains. We begin with a discussion of special types of hash function for which it is infeasible to compute the inverse function or find hash collisions. We show how these concepts extend to public-key encryption, which we first saw in Section 9.9. We then show how cryptographic hash functions can be used to ensure the anonymity, irrefutability, and tamper-resistance properties. We show how hash functions are used in mining algorithms later in Section 26.4.1.

26.3.1 Properties of Cryptographic Hash Functions

In Section 14.5, hash functions were used as a means of accessing data. Here, we use hash functions for a very different set of purposes, and as a result, we shall need hash functions with additional properties beyond those discussed earlier.

A hash function h takes input from some (large) domain of values and generates as its output a fixed-length bit string. Typically, the cardinality of the domain is much larger than the cardinality of the range. Furthermore, the hash function must have a *uniform distribution*, that is, each range value must be equally probable given random input. A hash function h is **collision resistant** if it is infeasible to find two distinct values x and y such that $h(x) = h(y)$. By **infeasible**, we mean that there is strong mathematical

[2]These properties pertain to blockchains, but not to most cryptocurrency exchanges. Most exchanges hold not only customers' data but also their keys, which means that a hack against the exchange's database can result in theft of users' private keys.

Telegram: @uni_k

www.konkur.in

evidence, if not an actual proof, that there is no way to find two distinct values x and y such that $h(x) = h(y)$ that is any better than random guessing.

The current standard choice of a cryptographic hash function is called SHA-256, a function that generates output 256 bits in length. This means that given a value x, the chance that a randomly chosen y will hash to the same value to which x hashes is $1/2^{256}$. This means that even using the fastest computers, the probability of a successful guess is effectively zero.[3]

The collision-resistance property contributes to the tamper resistance of a blockchain in a very important way. Suppose an adversary wishes to modify a block B. Since the next-newer block after B contains not only a pointer to B but also the hash of B, any modification to B must be such that the hash of B remains unchanged after the modification in order to avoid having to modify also that next-newer block. Finding such a modification is infeasible if the hash function has the collision-resistance property, and, therefore, any attempt to tamper with a block requires changing all newer blocks in the chain.

A second important property that we require of a cryptographic hash function is **irreversibility**, which means that given only $h(x)$, it is infeasible to find x. The term *irreversible* comes from the property that, given x, it is easy to compute $h(x)$, but given only $h(x)$, it is infeasible to find $h^{-1}(h(x))$. The next section shows how this concept is applied to blockchains.[4]

26.3.2 Public-Key Encryption, Digital Signatures, and Irrefutability

Section 9.9 described two categories of encryption methods: *private-key* encryption, where users share a secret key, and *public-key* encryption, where each user has two keys, a public key and a private key. The main problem with private-key encryption is that users must find a way at the outset to share the secret private key. Public-key encryption allows users who have never met to communicate securely. This property of public-key encryption is essential to blockchain applications that serve arbitrarily large communities of users worldwide.

Each user U_i has a public key E_i and a private key D_i. A message encrypted using E_i can be decrypted only with the key D_i, and, symmetrically, a message encrypted using D_i can be decrypted only with the key E_i, If user u_1 wishes to send a secure message x to U_2, U_1 encrypts x using the public key E_2 of user U_2. Only U_2 has the key D_2 to decrypt the result. For this to work, the specific function used must have the irreversibility property so that given a public key E_i it is infeasible to compute the inverse function,

[3] 2^{256} is larger than 10^{77}. If a computer could make one guess per cycle it would take more than 10^{67} seconds to have a 50 percent chance of guessing correctly. That translates to more than 10^{59} years. To put that in context, astronomers predict that the sun will have grown in size to envelop Earth within 10^{10} years.

[4] This property has long been used for storing passwords. Rather than storing user passwords in clear text, leaving them susceptible to being stolen, hashes are kept instead. Then, when a user logs in and enters a password, the hash of that password is computed and compared to the stored value. Were an attacker to steal the hashes, that attacker would still lack the actual passwords, and, if the hash function in use has the irreversibility property, then it is infeasible for the hacker to reverse-engineer the user passwords.

Telegram: @uni_k

www.konkur.in

that is, to find D_i. This creates a mechanism for users who have never met to share secret messages.

Suppose now that instead of seeking to send a secret message, user U_1 wishes to "sign" a document x. User U_1 can encrypt x using the private key D_1. Since this key is private, no one besides U_1 could have computed that value, but anyone can verify the signature by decrypting using the public key of U_1, that is, E_1. This provides a public proof that user U_1 has signed document x.

In blockchain applications, the concept of a digital signature is used to validate transactions. Observe that the linkage of blocks in the blockchain, using a pointer and the hash of block to which the pointer points, means that a user can sign an entire chain simply by signing the hash of the newest block in the chain. See the Further Reading section at the end of the chapter for references to the mathematics of public-key encryption.

26.3.3 Simple Blockchain Transactions

In our discussion of database transactions in Chapter 17, we described a transaction as a sequence of steps that read and/or write data values from the database. That concept of a transaction is based on a data model where there is a single store of data values that are accessed by transactions. A blockchain, in its simplest form, is more closely an analog of a database log in that it records the actual transactions and not just final data values. That analogy breaks down, however, in most blockchains, because transactions are either fully independent or depend explicitly on each other. The model we describe here corresponds to simple Bitcoin transactions.

As an example, consider two users, A and B, and assume A wishes to pay B 10 units of some currency. If this were a traditional banking application with a fiat currency such as the U.S. dollar, the transaction implementing this transfer would read A's account balance, decrement it by 10, and write that value to the database, and then read B's balance, add 10, and write that value to the database. In a blockchain-based system, this transaction is specified in a different manner.

Rather than referencing data items, a Bitcoin-style blockchain transaction references users and other transactions. Users are referenced by their user ID. User A would locate a transaction or set of transactions from past history $T_1, T_2, \ldots, T_n$ that paid A a total of at least 10 units of the currency. A would then create a transaction T that takes the output (i.e., the amount paid to A) by those transactions as input, and as its output pays 10 units of the currency to B and the remainder back to A as the "change." The original transactions $T_1, T_2, \ldots, T_n$ are then treated as having been spent.

Thus, each transaction indicates how much money has been paid to whom; the currency balance of a user A is defined by a set of unspent transactions that have paid money to A. Assuming A is honest, those transactions' outputs (i.e., the output of $T_1, T_2, \ldots, T_n$) would not have been spent already by A in a previous transaction. If A were indeed dishonest and T attempted to spend the output of some T_1 a second time, T would be a **double-spend** transaction. Double-spend transactions and other in-

Telegram: @uni_k

www.konkur.in

valid transactions are detected in the mining process that we discuss in Section 26.4, by keeping track of all unspent transactions and verifying that each transaction T_i that is input to T is unspent when T is executed. After T is executed, each such T_i is treated as spent.

Ethereum uses a different and more powerful model, where the blockchain maintains state (including current balance) for each account in the system. Transactions update the state, and can transfer funds from one account to another. The model used in Ethereum is discussed in Section 26.5.

A Bitcoin-style transaction T specifies:

- The input transactions $T_1, T_2, \ldots, T_n$.
- The set of users being paid and the amount to be paid to each, which in our example is 10 units to B and the remainder to A.[5]
- A's signature of the transaction, to prove that A in fact authorized this transaction.
- A more complex transaction might include executable code as part of its specification, but we shall defer that to Section 26.6.
- Data to be stored in the blockchain; the data must be under some size, which is blockchain dependent.

The transaction model described here is quite distinct from that of a traditional database system in a variety of ways, including:

- Existing data items are not modified. Instead, transactions add new information. As a result, not only the current state but also the history leading to the current state are fully visible.
- Conflicts in transaction ordering are prevented. If conflicts occur, the transaction causing a conflict is detected and deemed invalid as part of the process of adding a block to the chain, described in Section 26.4.
- Although the blockchain is a distributed system, a transaction is created locally. It becomes part of the permanent, shared blockchain only through the mining process. This is, in effect, a form of deferred transaction commit.
- Dependencies of one transaction upon another are stated explicitly in a transaction since a transaction lists those transactions whose outputs it uses as input. If we view this in terms of the precedence graph introduced in Chapter 17, our example would include precedence-graph edges $T_1 \rightarrow T, T_2 \rightarrow T, \ldots, T_n \rightarrow T$.
- There is no explicit concurrency control. Much of the need for concurrency control is eliminated by the maintenance of a complete history and the direct sequencing

[5] In a real system, there may also be a payout to the miner of the transaction, that is, the node that adds the block to the blockchain, as we discuss in Section 26.4.

Telegram: @uni_k

www.konkur.in

of transactions. Thus, there is no contention for the current value of any database data item.

This Bitcoin-based example is not the only way blockchain systems manage transaction ordering. We shall see another example when we consider smart contracts in Section 26.6.

The fact that data may be stored in the blockchain makes the blockchain more than just a tamper-resistant transaction log. It allows for the representation of any sort of information that might be stored in a traditional database. In Section 26.5.2, we shall see how this capability, particularly in blockchains with a concept of blockchain state, makes the blockchain a true database.

26.4 Consensus

Because the blockchain is replicated at all participating nodes, each time a new block is added, all nodes must eventually agree first on which node may propose a new block and then agree on the actual block itself.

In a traditional distributed database system, the consensus process is simplified by the fact that all participants are part of one controlling organization. Therefore, the distributed system can implement global concurrency control and enforce two-phase commit to decide on transaction commit or abort. In a blockchain, there may be no controlling organization, as is the case for a public blockchain like Bitcoin. In the case of a permissioned blockchain, there may be a desire to have a high degree of decentralized control in all matters except the actual permissioning of participants, which is managed by the organization controlling the permissioned blockchain.

When transactions are created, they are broadcast to the blockchain network. Nodes may collect a set of transactions to place in a new block to be added to the chain. The consensus mechanisms used in blockchains fall roughly into two categories:

1. Those where the nodes reach agreement on one node to add the next block. These typically use Byzantine consensus (Section 26.4.3).
2. Those where the blockchain is allowed temporarily to **fork** by allowing multiple nodes to create a block following the last block in the chain. In this approach, nodes attempt to add blocks to the longest linear subchain. Those blocks not on that longest chain are **orphaned** and not considered part of the blockchain. To avoid a massive number of forks being created, this approach limits the rate at which blocks may be added so that the expected length of orphaned branches is short. These typically use proof-of-work (Section 26.4.1) or proof-of-stake (Section 26.4.2).

A node that adds a block to the chain must first check that block of transactions. This entails checking that:

Telegram: @uni_k

www.konkur.in

- Each transaction is well-formed.
- The transaction is not double-spending by using as input (i.e., spending) currency units that have been used already by a prior transaction. To do so, each node must track the set of all unspent currency units (transactions), and look up this set for each transaction T to ensure that all the currency units that are inputs to T are unspent.
- The transaction is correctly signed by the submitting user.

When a node is selected to add a block to the chain, that block is propagated to all nodes, and each checks the block for validity before adding it to its local copy of the chain.

We next need to consider the question of why any node would want to use its resources for mining, that is to carry out the work needed to append blocks to the chain. Mining is a service to the blockchain network as a whole, and so miners are paid (in the currency of the blockchain) for their efforts. There are two sources of payment to miners:

1. A fee paid by the system in new coins in the currency of the blockchain.
2. A fee included by the submitter of the transaction. In this case the output of the transaction includes an additional output representing a payment to the miner of the block containing the transaction. Users are incented to include fees since such fees incent miners to include their transactions preferentially in new blocks.

The exact means of paying miners varies among blockchains.

In this section, we look at various ways to achieve consensus. We begin by assuming a public blockchain and describe consensus based on two approaches: *proof-of-work* and *proof-of-stake*. We then consider permissioned blockchains that in many cases choose to use a consensus mechanism based on *Byzantine consensus*.

26.4.1 Proof of Work

Proof-of-work consensus is designed for public blockchains in which the number of participating nodes is changing continuously. Any computer may download the blockchain and attempt to add blocks. As a result, a majority-based approach can be overwhelmed by an adversary who sets up a large number of low-cost computers as nodes. As mentioned earlier, such an attack is called a *Sybil attack*. Instead, proof-of-work requires a node to solve a computationally hard, but not infeasible, mathematical problem. An attacker cannot overwhelm a blockchain network simply by adding inexpensive nodes. Rather, the attacker would need to have access to computation capacity that forms a majority of the network's total computation capacity, a task that is much more difficult and costly than launching a Sybil attack.

Telegram: @uni_k

www.konkur.in

The computationally hard problem is based on the concept of cryptographic hashing. A node that wishes to mine a block B as the next block needs to find a value, called a **nonce**, that, when concatenated to B and the hash of the previous block, hashes to a value less than a preset target value specified for the blockchain. The nonce is typically a 32-bit value. If the target is set very low, say to 4, and assuming the usual 256-bit hash, a miner would have only a $1/2^{254}$ chance of succeeding for a single choice for the nonce. If the target were set very high, say to 2^{255}, the miner would have a 50 percent chance of success. Blockchain implementations are designed to vary the target so as to control the rate of mining of blocks across the whole system. This variability allows the system to adjust as computation power increases whether due to hardware advances or due to additional nodes joining the network. The target times vary for different blockchains. Bitcoin targets having some node in the system successfully mine a block every 10 minutes. Ethereum targeted a mining time of 10 to 15 seconds with its proof-of-work mechanism. As of late 2018, Ethereum is moving to a proof-of-stake mechanism and is expected to target a slightly faster rate. While faster may appear to be better, note that if mining occurs at a faster rate than the time it takes to propagate a new block throughout the network, the probability of forks and orphaned blocks increases.

Now that we have seen how proof-of-work mining works, let us recall the properties we stated about cryptographic hash functions. If there were an efficient algorithm for finding a nonce that results in a hash less than the target, miners might find nonces too quickly. Therefore, the hash function must ensure that there is no better way to find a nonce than simply trying each possible nonce value in turn. This leads us to require one additional property for cryptographic hash functions, the **puzzle-friendliness** property. This property requires that given a value k, for any n-bit value y it is infeasible to find a value x such that $h(x\|k) = y$ in time significantly less that 2^n, where $\|$ denotes concatenation of bit strings.

Proof-of-work mining is controversial. On the positive side, for a large network, it would be highly costly for an adversary to obtain enough computational power to dominate mining. However, on the negative side, the amount of energy used in mining is huge. Estimates as this chapter is being written suggest that Bitcoin mining worldwide consumes about 1 percent of the power consumed by the United States, or more than the entire consumption of several nations, for example Ireland. The large amount of computation needed has created incentives to design special-purpose computing chips for mining and incentives to locate large mining installations near sources of cheap power sources.

These concerns are causing a movement to alternatives, such as proof-of-stake, which we discuss next. These concerns have led also to interest in alternative forms of proof-of-work that, for example, require having a large amount of main memory in order quickly to find a nonce. Memory-intensive schemes retain the cost barrier of proof-of-work while reducing the energy waste. They are a subject of current research. Furthermore, we shall see that for enterprise permissioned-blockchain applications, much less costly means of consensus are possible.

Telegram: @uni_k

www.konkur.in

In practice, a group of users may unite to form a *mining pool*, which is a consortium that works together to mine blocks and then shares the proceeds among its members.

26.4.2 Proof of Stake

The concept of proof-of-stake is to allow nodes holding a large stake in the currency of the blockchain to be chosen preferentially to add blocks. This rule cannot be applied absolutely, since then a single largest stakeholder would control the chain. Instead, the probability of mining success, using proof-of-work, is made higher for nodes in proportion to their stake. By adjusting both the stake requirements and the mining difficulty, it remains possible to control the rate at which blocks are mined.

There are a wide variety of proof-of-stake schemes. They may include measurement not only of overall stake, but also the total time a stake has been held. They may require that the stake or some fraction of it be held inactive for some period of time in the future.

Properly tuning a proof-of-stake mechanism is difficult. Not only are there more parameters to consider than in proof-of-work, but also one must guard against a situation where there is too little cost penalty for an adversary to add blocks to a fork other than the longest one.

26.4.3 Byzantine Consensus

An important alternative to work- or stake-based consensus is message-based consensus. Message-based consensus is widely used in distributed database systems. As we noted earlier, the basic consensus protocols do not work for blockchain consensus because it cannot be assumed that there are no malicious nodes.

Message-based systems aim to achieve consensus via a majority vote. Such systems are vulnerable to a Sybil attack. In an enterprise permissioned blockchain, in which users have to be granted permission to participate, Sybil attacks are not possible since the permissioning authority can easily deny permission when a malicious user attempts to add an excessive number of nodes. However, even in this setting, one cannot assume every user is totally honest.

For example, consider a supply-chain blockchain in which all suppliers enter data on the chain pertaining to each item being supplied either to another supplier or the ultimate manufacturer of an end-user product. Some supplier might choose to falsify data for its own advantage, but, when a fraud investigation begins, that supplier may then seek to fork the blockchain to cover-up its fraud. Thus, even absent the possibility of Sybil attacks, there remains the possibility of adversarial behavior. It is difficult to anticipate every possible form of adversarial behavior.

For this reason, we model this situation using the concept of **Byzantine failure** in which it is assumed that a "failed" node can behave in an arbitrary manner, and the network of non-failed nodes must be robust to all such misbehavior, including misbehavior that takes exactly the needed set of steps to sabotage the network. The assumption of

Telegram: @uni_k

www.konkur.in

Byzantine failure is quite different from the assumption made by consensus protocols, where the only type of failure considered is the absence of function, that is, the only way a node or network link fails to stop working and thus do nothing. This is referred to as a *fail-stop* model and precludes any malicious behavior.

In Section 23.8, we discussed distributed consensus protocols, notably Paxos and Raft. These protocols depend on the fail-stop assumption, but allow agreement using majority rule (in contrast, 2PC requires unanimity of agreement). For Byzantine consensus, we must seek a form of majority rule that overcomes not only the failure of a minority of nodes, but also the possible malicious behavior of that minority. For example, a malicious node n_1 may tell node n_2 that it desires to commit a transaction, but tell n_3 that it desires to abort the transaction. As one might expect, achieving consensus in the face of such malicious nodes requires a higher cost in the number of messages sent to achieve agreement, but in a blockchain, that higher cost is acceptable since it can be much lower than the cost of proof-of-work or proof-of-stake mining.

The development of Byzantine consensus algorithms began in the early 1980s; see the Further Reading section at the end of the chapter for references. There has been much theoretical work relating the number of rounds of messaging, the total number of messages sent, and the fraction of the nodes that can be malicious without causing the protocol to fail. Early work made assumptions about network behavior, such as the time it takes to deliver a message or that the network behaves in a highly synchronous manner. Modern Byzantine consensus algorithms are based on real-world assumptions and incorporate cryptographic signatures to guard against forged messages. The degree of synchronization is reduced, but truly asychronous fault-tolerant consensus is provably impossible. One widely used approach, called *Practical Byzantine Fault Tolerance*, tolerates malicious failure of up to $\lfloor \frac{n-1}{3} \rfloor$ nodes and is viewed as providing an acceptable level of performance. Other protocols are referenced in the Further Reading section at the end of the chapter.

26.5 Data Management in a Blockchain

Until now we have not been concerned about the efficiency of looking up information in a blockchain. While individual users can track their unspent transactions, that is not sufficient to validate a block. Each node needs to be able to check each transaction in a block to see if it was already spent. In principle, that could be done by searching the entire blockchain, but that is far too costly since it could involve searching backwards to the very first block in the chain. In this section, we shall consider data structures to make such lookups efficient.

Furthermore, not every blockchain uses a transaction model in which transaction inputs are restricted to be the direct output of other transactions. Some, notably Ethereum, allow for the maintenance of a state for each user (account, in Ethereum parlance) that holds the account balance (in the Ethereum currency, Ether) and some

Telegram: @uni_k

www.konkur.in

associated storage. This transaction and data model comes closer to that of a database system. Simply storing this information in a database, however, would not preserve the blockchain properties we listed in Section 26.2. In this section, we consider this richer model and how it can be represented physically either via specialized data structures or with the help of database system concepts.

26.5.1 Efficient Lookup in a Blockchain

As we noted earlier, in order to validate a Bitcoin-style transaction, a node needs to check three items:

1. The transaction is syntactically well formed (proper data format, sum of inputs equals sum of outputs, and so on). This is relatively straightforward.
2. The transaction is signed by the user submitting it. This is a matter of ensuring that the signature, which should have been produced by the user submitting the transaction using her or his private key, can be decrypted with that user's public key to obtain the transaction itself. This is not a highly costly step.
3. The transaction's inputs have not been spent already. This entails looking up each individual input transaction in the blockchain. These transactions could be anywhere in the blockchain since they can be arbitrarily old. Without a good means of performing this lookup, this step would be prohibitively costly.

To test for an input transaction having been used already, it is necessary to be able to check that transaction did not appear earlier as input to another transaction. Thus, it suffices for each node to maintain an index on all unspent transactions. Entries in this index point to the location of the corresponding transaction in the blockchain, allowing the details of the input transaction to be validated.

Bitcoin, like many other blockchains, facilitates lookup and validation by storing transactions within a block in a Merkle tree, which we discussed in Section 23.6.6. In that section, we noted that a Merkle tree enables the efficient verification of a collection (transactions, in the case of a blockchain) that may have been corrupted by a malicious user. In a blockchain, there are optimizations to the Merkle tree possible, such as truncating the tree to remove subtrees consisting solely of spent transactions. This reduces significantly the space requirements for nodes to store the full blockchain. Space is a major consideration since major blockchains grow faster that one gigabyte per month, a rate likely to increase as blockchain applications grow.

The Merkle-tree structure is particularly useful for light nodes (i.e., nodes that do not store the entire blockchain) since they need to retain only the root hash of the tree for verification. A full node can then provide any needed data to the light node by providing those data plus the hashes needed for the light node to verify that the provided data are consistent with its stored hash value (see Section 23.6.6).

Telegram: @uni_k

www.konkur.in

26.5.2 Maintaining Blockchain State

The simple blockchain transaction model of Section 26.3.3 showed how a basic Bitcoin transaction works. There are more complex transactions possible in Bitcoin, but they follow the same pattern of a set of input transactions and a set of payments to users.

In this section, we look at the model used by certain other blockchains, notably Ethereum, that maintain a *state* that holds the balance in each account. Transactions move currency units (*ether* in Ethereum) among accounts. Since transactions are serialized into blocks by miners, there is no need for concurrency-control protocols like those of Chapter 18. Each block contains a sequence of transactions but also contains the state as it existed after execution of transactions in the block. It would be wasteful to replicate the entire state in each block since the modest number of transactions in one block are likely to change a relatively small fraction of the overall state. This creates a need for a data structure allowing better use of storage.

Recall that transactions within a block are stored in a Merkle tree. State is stored similarly. This would appear to offer the possibility of saving space by allowing pointers (plus the associated hash) back to earlier blocks for those parts of the state that are unchanged. The only challenge here is that it must be possible not only to change tree nodes, but also to insert and delete them. A variant of the Merkle-tree data structure, called a **Merkle-Patricia tree**, is used for this purpose in some blockchains, including Ethereum. This data structure allows for efficient key-based search in the tree. Instead of actually deleting and inserting tree nodes, a new tree root is created and the tree itself structured so as to reference (and thus reuse) subtrees of prior trees. Those prior trees are immutable, so rather than making new parent pointer (which we can't do), a leaf-to-root path is generated by reversing a root-to-leaf path that is easily obtained in the Merkle-Patricia tree structure. Details of this data structure can be found in the references in the Further Reading section at the end of the chapter.

Corda, Hyperledger Fabric, and BigchainDB are examples of blockchains that use a database to store state and allow querying of that state. Fabric and BigchainDB use NoSQL databases. Corda uses an embedded-SQL database. In contrast, Ethereum state is stored in a key-value store.

26.6 Smart Contracts

So far, we have focused on simple funds-transfer transactions. Actual blockchain transactions can be more complex because they may include executable code. Blockchains differ not only in the supported language(s) for such code, but also, and more importantly, in the power of those languages. Some blockchains offer Turing-complete languages, that is, languages that can express all possible computations. Others offer more limited languages.

26.6.1 Languages and Transactions

Bitcoin uses a language of limited power that is suitable for defining many standard types of conditional funds-transfer transactions.

Telegram: @uni_k

www.konkur.in

instruction, which requires *m* of *n* specified users to approve the transfer. This enables escrow transactions in which a trusted third party resolves any dispute between the two parties to the actual transfer. It also enables grouping several transactions between two users into one larger transaction without having to submit each component transaction separately to the blockchain. Because adding transactions to the blockchain has a time delay and a cost in transaction fees, this feature is quite important. This concept has been extended in off-chain processing systems, which we discuss in Section 26.7.

Ethereum as well as most blockchains targeting enterprise applications include a language that is Turing complete. Many use familiar programming languages or variants based heavily on such languages. This would seem like an obvious advantage over less-powerful languages, but it comes at some risk. Whereas it is impossible to write an infinite loop in Bitcoin's language, it is possible to do so in any Turing-complete language. A malicious user could submit a transaction that encodes an infinite loop, thereby consuming an arbitrarily large amount of resources for any node attempting to include that transaction in a newly mined block. Testing code for termination, the halting problem, is a provably unsolvable problem in the general case. Even if the malicious user avoids an infinite loop, that user could submit code that runs for an exceptionally long time, again consuming miner resources. The solution to this problem is that users submitting a transaction agree to pay the miner for code execution, with an upper bound placed on the payment. This limits the amount of total execution to some bounded amount of time.

The decentralized nature of a blockchain leads to an incentive system for users to convince miners to include their transaction and thus execute their code. Ethereum's solution is based on the concept of *gas*, so named as to provide an analogy to automobile fuel. Each instruction consumes a fixed amount of gas. Gas consumption in a transaction is governed by three parameters:

1. **Gas price:** the amount of ether the user is offering to pay the miner for one unit of gas.
2. **Transaction gas limit:** the upper bound on transaction gas consumption. Transactions that exceed their gas limit are aborted. The miner keeps the payment, but the transaction actions are never committed to the blockchain.
3. **Block gas limit:** a limit in the blockchain system itself on the sum over all transactions in a block of their transaction gas limits.

A user who sets a gas price too low may face a long wait to find a miner willing to include the transaction. Setting the gas price too high results in the user overpaying.

Another hard choice is that of the gas limit. It is hard to set the limit to the precise amount of gas that the contract will use. A user who sets the limit too low risks transaction failure, while a user who, fearing "running out of gas," sets the transaction gas limit excessively high may find that miners are unwilling to include the transaction because it consumes too large a fraction of the block gas limit. The result of this is an

Telegram: @uni_k

www.konkur.in

interesting problem for transaction designers in optimizing for both cost and speed of mining.

In a Bitcoin-style transaction, transaction ordering is explicit. In a state-based blockchain like Ethereum, there is no explicit concept of input transactions. However, there may be important reasons why a smart contract may wish to enforce a transaction order. For transactions coming from the same account, Ethereum forces those transactions to be mined in the order in which the account created them by means of an *account nonce* associated with the transaction. An account nonce is merely a sequence number associated with each transaction from an account, and the set of transactions from an account must have consecutive sequence numbers. Two transactions from an account cannot have the same sequence number, and a transaction is accepted only after the transaction with the previous sequence number has been accepted, thus preventing any cheating in transaction ordering. If the transactions to be ordered are from different accounts, they need to be designed such that the second transaction in the ordering would fail to validate until after the first transaction is processed.

The fact that miners must run the smart-contract code of transactions they wish to include in a block, and that all full nodes must run the code of all transactions in mined blocks, regardless of which node mined the block, leads to a concern about security. Code is run in a safe manner, usually on a virtual machine designed in the style of the Java virtual machine. Ethereum has its own virtual machine, called the EVM. Hyperledger executes code in Docker containers.

26.6.2 External Input

A smart contract may be defined in terms of external events. As a simple example, consider a crop-insurance smart contract for a farmer that pays the farmer an amount of money dependent on the amount of rainfall in the growing season. Since the amount of rainfall in any future season is not known when the smart contract is written, that value cannot be hard-coded. Instead, input must be taken from an external source that is trusted by all parties to the smart contract. Such an external source is called an **oracle**.[6]

Oracles are essential to smart contracts in many business applications. The fact that the oracle must be trusted is a compromise on the general trustlessness of a blockchain environment. However, this is not a serious compromise in the sense that only the parties to a contract need to agree on any oracles used and, once that agreement is made, the agreement is coded into the smart contract and is immutable from that point forward.

Corruption of an oracle after it is coded into an operating smart contract is a real problem. This issue could be left as an externality for the legal system but ideally, a process for settlement of future disputes would be coded into the contract in a variety of ways. For example, parties to the contract could be required to send the contract

[6]This term is rooted in ancient Greek culture and bears no

Telegram: @uni_k

www.konkur.in

certification messages periodically, and code could be written defining actions to be taken in case a party fails to recertify its approval of the oracle.

Direct external output from a smart contract is problematic since such output would have to occur during its execution and thus before the corresponding transaction is added to the blockchain. Ethereum, for example, deals with this by allowing a smart contract to *emit events* that are then logged in the blockchain. The public visibility of the blockchain then allows the actions of the smart contract to trigger activity external to the blockchain.

26.6.3 Autonomous Smart Contracts

In many blockchains, including Ethereum, smart contracts can be deployed as independent entities. Such smart contracts have their own account, balance, and storage. This allows users (or other smart contracts) to use services provided by a smart contract and to send or receive currency from a smart contract.

Depending on how a specific smart contract is coded, a user may be able, by design, to control the smart contract by sending it messages (transactions). A smart contract may be coded so that it operates indefinitely and autonomously. Such a contract is referred to as a **distributed autonomous organization** (DAO).[7] DAOs, once established, are difficult to control and manage. There is no way to install bug fixes. In addition, there are many unanswered questions about legal and regulatory matters. However, the ability to create these entities that communicate, store data, and do business independent of any user is one of the most powerful features of the blockchain concept. In an enterprise setting, smart contracts operate under some form of control by an organization or a consortium.

A smart contract may be used to create a currency on top of another currency. Ethereum often serves as the base blockchain as this allows the rich existing ecosystem for Ethereum to be leveraged to provide underlying infrastructure. Such higher-level currency units are called tokens, and the process of creating such currencies is referred to as an **initial coin offering** (ICO). An important added benefit of using an existing blockchain as the basis for a token is that it is then possible to reuse key elements of the user infrastructure, most importantly the wallet software users need to store tokens. The ERC-20 Ethereum standard for tokens is widely used. More recent standards, including ERC-223, ERC-621, ERC-721, ERC-777, and ERC-827, are discussed in the references in the Further Reading section at the end of the chapter.

The relative ease of creating an ICO has made it an important method of funding new ventures, but this has also led to several scams, resulting in attempts by governments to regulate this fundraising methodology.

Beyond fundraising, an important application of smart contracts is to create independent, autonomous service providers whose operation is controlled not by humans

[7]The general use of DAO is distinct from a specific distributed autonomous organization called "The DAO". The DAO was a crowdfunded venture-capital operation that failed due to a bug that enabled a major theft of funds (see Note 26.1 on page 1258).

Telegram: @uni_k

www.konkur.in

but by source code, often open-source. In this way, trustless services that do not require their users to trust any person or organization can be created. As we noted earlier, a fully autonomous contract cannot be stopped or modified. Thus, bugs last forever, and the contract can continue as long as it can raise enough currency to support its operation (i.e., for Ethereum, earn enough ether to pay for gas). These risks suggest that some compromise on the concept of trustlessness may make sense in smart-contract design, such as giving the contract creator the ability to send a self-destruct message to the contract.

26.6.4 Cross-Chain Transactions

Up to this point, we have assumed implicitly that a blockchain transaction is limited to one specific blockchain. If one wished to transfer currency from an account on one blockchain to another account that is on a different blockchain, not only is there the issue that the currencies are not the same, but also there is the problem that the two blockchains have to agree on the state of this cross-chain transaction at each point in time.

We have seen a related problem for distributed databases. If a single organization controls the entire distributed system, then two-phase commit can be used. However, if the system is controlled by multiple organizations as in the federated systems discussed in Section 23.5.3, coordination is more difficult. In the blockchain setting, the high level of autonomy of each system and the requirement of immutability set an even higher barrier.

The simplest solution is to use a trusted intermediary organization that operates much like one that exchanges traditional fiat currencies.

If both users have accounts on both blockchains, a trustless transaction can be defined by creating transactions on each chain for the required funds transfer that are designed such that if one transaction is added to its blockchain its smart-contract code reveals a secret that ensures that other transactions cannot be canceled. Techniques used include the following, among others:

- Time-lock transactions that reverse after a certain period of time unless specific events occur.
- Cross-chain exchange of Merkle-tree headers for validation purposes.

A risk in these techniques is the possibility that a successfully mined transaction winds up on an orphaned fork, though there are ways to mitigate these risks. The details are system specific. See the Further Reading section at the end of the chapter.

A more general solution is to create a smart contract that implements a market similar conceptually to a stock exchange in which willing buyers and sellers are matched. Such a contract operates in the role of trusted intermediary rather than a human-run bank or brokerage as would be used for fiat currencies. The technical issues in cross-chain transactions remain an area of active research.

Telegram: @uni_k

www.konkur.in

26.7 Performance Enhancement

At a high level, a blockchain system may be viewed as having three major components:

1. **Consensus management:** Proof-of-work, proof-of-stake, Byzantine consensus, or some hybrid approach. Transaction processing performance is dominated by the performance of consensus management.
2. **State-access management:** Access methods to retrieve current blockchain state, ranging from a simple index to locate transactions from a specific account-id or user ID, to key-value store systems, to a full SQL interface.
3. **Smart contract execution:** The environment that runs the (possibly compiled) smart-contract code, typically in a virtualized environment for security and safety.

The rate of transaction processing, referred to as throughput, in blockchain systems is significantly lower than in traditional database systems. Traditional database systems are able to process simple funds-transfer transactions at peak rates on the order of tens of thousands of transactions per second. Blockchain systems' rates are less; Bitcoin processes less than 10 per second, and Ethereum, at present, only slightly more than 10 per second.[8] The reason is that techniques such as proof-of-work limit the number of blocks that can be added to the chain per unit time, with Bitcoin targeting one block every 10 minutes. A block may contain multiple transactions, so the transaction processing rate is significantly more than 1 in 10 minutes, but is nevertheless limited.

In most applications, transaction throughput is not the only performance metric. A second and often more important metric is transaction latency, or response time. Here, the distributed consensus required by blockchain systems presents a serious problem. As an example, we consider Bitcoin's design in which the mining rate is maintained close to 1 block every 10 minutes. That alone creates significant latency, but added to that is the need to wait for several subsequent blocks to be mined so as to reduce the probability that a fork will cause the transaction's block to be orphaned. Using the usual recommendation of waiting for 6 blocks, we get a true latency of 1 hour. Such response times are unacceptable for interactive, real-time transaction processing. In contrast, traditional database systems commit individual transactions and can easily achieve millisecond response time.

These transaction processing performance issues are primarily issues due to consensus overhead with public blockchains. Permissioned blockchains are able to use faster message-based Byzantine consensus algorithms, but other performance issues still remain, and are continuing to be addressed.

[8] At the time of publication, Ethereum's architects are contemplating advocating a fork to allow faster, lower-overhead mining.

Telegram: @uni_k

www.konkur.in

26.7.1 Enhancing Consensus Performance

There are two primary approaches to improve the performance of blockchain consensus:

1. **Sharding:** distributing the task of mining new blocks to enable parallelism among nodes.
2. **Off-chain transaction processing:** Trusted systems that process transactions internally without putting them on the blockchain. These transactions are grouped into a single transaction that is then placed on the blockchain. This grouping may occur with some agreed-upon periodicity or occur only at the termination of the agreement.

Sharding is the partitioning of the accounts in a blockchain into shards that are mined separately in parallel. In the case where a transaction spans shards, a separate transaction is run on each shard with a special system-internal cross-shard transaction recorded to ensure that both parts of the given transaction are committed. The overhead of the cross-shard transaction is low. There are some risks resulting from the fact that splitting the mining nodes up by shard results in smaller sets of miners that are then more vulnerable to attack since the cost to attack a smaller set of miners is less. However, there are ways to mitigate this risk.

Off-chain transactions require deployment of a separate system to manage those transactions. The best known of these is the Lightning network, which not only speeds blockchain transactions via off-chain processing but also can process certain cross-chain transactions. Lightning promises transaction throughput and latency at traditional database-system rates, but provides this at the cost of some degree of anonymity and immutability (i.e., transactions that commit off-chain, but are rejected at the blockchain). By increasing the frequency of transaction confirmations to the blockchain, one can decrease the loss of immutability at the price of reduced performance improvement.

26.7.2 Enhancing Query Performance

Some blockchain systems offer little more than an index on user or account identifiers to facilitate looking up unspent transactions. This suffices for a simple funds-transfer transaction. Complex smart contracts, however, may need to execute general-purpose queries against the stored current state of the blockchain. Such queries may perform the equivalent of join queries, whose optimization we studied at length in Chapter 16. However, the structure of blockchain systems, in which state-access management may be separate from the execution engine may limit the use of database-style query optimization. Furthermore, the data structures used for state representation, such as the Merkle-Patricia tree structure we saw in Section 26.5.2, may limit the choice of algorithms to implement join-style queries.

Telegram: @uni_k

www.konkur.in

Blockchain systems built on a traditional or a NoSQL database keep state information within that database and allow smart contracts to run higher-level database-style queries against that state. Those advantages come at the cost of using a database-storage format that may lack the rigorous cryptographic protection of a true blockchain. A good compromise is for the database to be hosted by a trusted provider with updates going not only to the database but also to the blockchain, thus enabling any user who so wishes to validate the database against the secure blockchain.

26.7.3 Fault-Tolerance and Scalability

Performance in the presence of failures is a critical aspect of a blockchain system. In traditional database systems, this is measured by the performance of the recovery manager and, as we saw in Section 19.9, the ARIES recovery algorithm is designed to optimize recovery time. A blockchain system, in contrast, uses a consensus mechanism and a replication strategy designed for continuous operation during failures and malicious attacks, though perhaps with lower performance during such periods. Therefore, besides measuring throughput and latency, one must also measure how these performance statistics change in times of failure or attack.

Scalability is a performance concern in any distributed system as we saw in Chapter 20. The architectural differences between blockchain systems and parallel or distributed database systems introduce challenges in both the measure of scaleup and its optimization. We illustrate the differences by considering the relative scalability of 2PC and Byzantine consensus. In 2PC, a transaction accessing a fixed number of nodes, say five, needs only the agreement of these five nodes, regardless of the number of nodes in the system. If we scale the system up to more nodes, that transaction still needs only those five nodes to agree (unless the scaling added a replica site). Under Byzantine consensus, every transaction needs the agreement of a majority of the non-failed nodes, and so, the number of nodes that must agree not only starts much larger but also grows faster as the network scales.

The Further Reading section at the end of the chapter provides references that deal with the emerging issue of blockchain performance measurement and optimization.

26.8 Emerging Applications

Having seen how blockchains work and the benefits they offer, we can look at areas where blockchain technology is currently in use or may be used in the near future.

Applications most likely to benefit from the use of a blockchain are those that have high-value data, including possibly historical data, that need to be kept safe from malicious modification. Updates would consist mostly of appends in such applications. Another class of applications that are likely to benefit area those that involve multiple cooperating parties, who trust each other to some extent, but not fully, and desire to have a shared record of transactions that are digitally signed, and are kept safe from tampering. In this latter case, the cooperating parties could include the general public.

Telegram: @uni_k

www.konkur.in

Below, we provide a list of several application domains along with a short explanation of the value provided by a blockchain implementation of the application. In some cases, the value added by a blockchain is a novel capability; in others, the value added is the ability to do something that could have been done previously only at prohibitive cost.

- **Academic certificates and transcripts:** Universities can put student certificates and transcripts on a public blockchain secured by the student's public key and signed digitally by the university. Only the student can read the records, but the student can then authorize access to those records. As a result, students can obtain certificates and transcripts for future study or for prospective employers in a secure manner from a public source. This approach was prototyped by MIT in 2017.
- **Accounting and audit:** Double-entry bookkeeping is a fundamental principle of accounting that helps ensure accurate and auditable records. A similar benefit can be gained from cryptographically signed blockchain entries in a digital distributed ledger. In particular, the use of a blockchain ensures that the ledger is tamperproof, even against insider attacks and hackers who may gain control of the database. Also, if the enterprise's auditor is a participant, then ledger entries can become visible immediately to auditors, enabling a continuous rather than periodic audit.
- **Asset management:** Tracking ownership records on a blockchain enables verifiable access to ownership records and secure, signed updates. As an example, real-estate ownership records, a matter of public record, could be made accessible to the public on a blockchain, while updates to those records could be made only by transactions signed by the parties to the transaction. A similar approach can be applied to ownership of financial assets such as stocks and bonds. While stock exchanges manage trading of stocks and bonds, long term records are kept by depositories that users must trust. Blockchain can help track such assets without having to trust a depository.
- **E-government:** A single government blockchain would eliminate agency duplication of records and create a common, authoritative information source. A highly notable user of this approach is the government of Estonia, which uses its blockchain for taxation, voting, health, and an innovative "e-Residency" program.
- **Foreign-currency exchange:** International financial transactions are often slow and costly. Use of an intermediary cryptocurrency can enable blockchain-based foreign-currency exchange at a relative rapid pace with full, irrefutable traceability. Ripple is offering such capability using the XRP currency.
- **Health care:** Health records are notorious for their nonavailability across healthcare providers, their inconsistency, and their inaccuracy even with the increased use of electronic health records. Data are added from a large number of sources and the provenance of materials used may not be well documented (see discussion of supply chains below). A unified blockchain is suitable for distributed update,

Telegram: @uni_k

www.konkur.in

and cryptographic data protection, unlockable by the patient's private key, would enable access to a patient's full health record anytime, anywhere in an emergency. The actual records may be kept offchain, but the blockchain acts as the trusted mechanism for accessing the data.

- **Insurance claims:** The processing of insurance claims is a complex workflow of data from the scene of the claim, various contractors involved in repairs, statements from witnesses, etc. A blockchain's ability to capture data from many sources and distribute it rapidly, accurately, and securely, promises efficiency and accuracy gains in the insurance industry.
- **Internet of things:** The Internet of Things (IoT) is a term that refers to systems of many interacting devices ("things"), including within smart buildings, smart cities, self-monitoring civil infrastructure, and so on. These devices could act as nodes that can pass blockchain transactions into the network without having to ensure the transmission of data reaches a central server. In the late 2010s, research is underway to see if this data-collection approach can be effective in lowering costs and increasing performance. Adding so many entries to a blockchain in a short period of time may suggest a replacement of the chain data structure with a directed, acyclic graph. The Iota blockchain is an example of one such system, where the graph structure is called a *tangle*.
- **Loyalty programs and aggregation of transactions:** There are a variety of situations where a customer or user makes multiple purchases from the same vendor, such as within a theme park, inside a video game, or from a large online retailer. These vendors could create internal cryptocurrencies in a proprietary, permissioned blockchain, with currency value pegged to a fiat currency like the dollar. The vendor gains by replacing credit-card transactions with vendor-internal transactions. This saves credit-card fees and allows the vendor to capture more of the valuable customer data coming from these transactions. The same concept can apply to retail loyalty points, exemplified by airline frequent-flyer miles. It is costly for vendors to maintain these systems and coordinate with partner vendors in the program. A blockchain-based system allows the hosting vendor to distribute the workload among the partners and allows transactions to be posted in a decentralized manner, releaving the vendor from have to run its own online transaction processing system. In the late 2010s, business strategies were being tested around these concepts.
- **Supply chain:** Blockchain enables every participant in a supply chain to log every action. This facilitates tracking the movement of every item in the chain rather than only aggregates like crates, shipments, etc. In the event of a recall, the set of affected products can be pinpointed to a smaller set of products and done so quickly. When a quality issue suggests a recall, some supply-chain members may be tempted to cover up their role, but the immutability of the blockchain prevents record falsification after the fact.

Telegram: @uni_k

www.konkur.in

- **Tickets for events:** Suppose a person A has bought tickets for an event, but now wishes to sell them, and B buys the ticket from A. Given that tickets are all sold online, B would need to trust that the ticket given by A is genuine, and A has not already sold the ticket, that is, the ticket has not been double-spent. If ticket transactions are carried out on a blockchain, double-spending can be detected easily. Tickets can be verified if they are signed digitally by the event organizer (whether or not they are on a blockchain).
- **Trade finance:** Companies often depend on loans from banks, issued through letters of credits, to finance purchases. Such letters of credit are issued against goods based on bills of lading indicating that the goods are ready for shipment. The ownership of the goods (title) is then transferred to the buyer. These transactions involve multiple parties including the seller, buyer, the buyer's bank, the seller's bank, a shipping company and so forth, which trust each other to some extent, but not fully. Traditionally, these processes were based on physical documents that have to be signed and shipped between parties that may be anywhere on the globe, resulting in significant delays in these processes. Blockchain technology can be used to keep these documents in a digital form, and automate these processes in a way that is highly secure yet very fast (at least compared to processing of physical documents).

Other applications beyond those we have listed continue to emerge.

26.9 Summary

- Blockchains provide a degree of privacy, anonymity, and decentralization that is hard to achieve with a traditional database.
- Public blockchains are accessibly openly on the internet. Permissioned blockchains are managed by an organization and usually serve a specific enterprise or group of enterprises.
- The main consensus mechanisms for public blockchains are proof-of-work and proof-of-stake. Miners compete to add the next block to the blockchain in exchange for a reward of blockchain currency.
- Many permissioned blockchains use a Byzantine consensus algorithm to choose the node to add the next block to the chain.
- Nodes adding a block to the chain first validate the block. Then all full nodes maintaining a replica of the chain validate the new block.
- Key blockchain properties include irrefutability and tamper resistance.
- Cryptographic hash functions must exhibit collision resistance, irreversibility, and puzzle friendliness.

Telegram: @uni_k

www.konkur.in

- Public-key encryption is based on a user having both a public and private key to enable both the encryption of data and the digital signature of documents.
- Proof-of-work requires a large amount of computation to guess a successful nonce that allows the hash target to be met. Proof-of-stake is based on ownership of blockchain currency. Hybrid schemes are possible.
- Smart contracts are executable pieces of code in a blockchain. In some chains, they may operate as independent entities with their own data and account. Smart contracts may encode complex business agreements and they may provide ongoing services to nodes participating in the blockchain.
- Smart contracts get input from the outside world via trusted oracles that serve as a real-time data source.
- Blockchains that retain state can serve in a manner similar to a database system and may benefit from the use of database indexing methods and access optimization, but the blockchain structure may place limits on this.

Review Terms

- Public and permissioned blockchain
- Cryptographic hash
- Mining
- Light and full nodes
- Proof-of-work
- Proof-of-stake
- Byzantine consensus
- Tamper resistance
- Collision resistance
- Irreversibility
- Public-key encryption
- Digital signature
- Irrefutability
- Forks: hard and soft
- Double spend
- Orphaned block
- Nonce
- Block validation
- Merkle tree
- Patricia tree
- Bitcoin
- Ethereum
- Gas
- Smart contract
- Oracles
- Cross-chain transaction
- Sharding
- Off-chain processing

Practice Exercises

26.1 What is a blockchain fork? List the two types of fork and explain their differences.

Telegram: @uni_k

www.konkur.in

26.2 Consider a hash function $h(x) = x \bmod 2^{256}$, that is, the hash function returns the last 256 bits of x.
Does this function have

a. collision resistance

b. irreversibility

c. puzzle friendliness

Why or why not?

26.3 If you were designing a new public blockchain, why might you choose proof-of-stake rather than proof-of-work?

26.4 If you were designing a new public blockchain, why might you choose proof-of-work rather than proof-of-stake?

26.5 Explain the distinction between a public and a permissioned blockchain and when each would be more desirable.

26.6 Data stored in a blockchain are protected by the tamper-resistance property of a blockchain. In what way is this tamper resistance more secure in practice than the security provided by a traditional enterprise database system?

26.7 In a public blockchain, how might someone determine the real-world identity that corresponds to a given user ID?

26.8 What is the purpose of *gas* in Ethereum?

26.9 Suppose we are in an environment where users can be assumed not to be malicious. In that case, what advantages, if any, does Byzantine consensus have over 2PC?

26.10 Explain the benefits and potential risks of sharding.

26.11 Why do enterprise blockchains often incorporate database-style access?

Exercises

26.12 In what order are blockchain transactions serialized?

26.13 Since blockchains are immutable, how is a transaction abort implemented so as not to violate immutability?

26.14 Since pointers in a blockchain include a cryptographic hash of the previous block, why is there the additional need for replication of the blockchain to ensure immutability?

26.15 Suppose a user forgets or loses her or his private key? How is the user affected?

Telegram: @uni_k

www.konkur.in

26.16 How is the difficulty of proof-of-work mining adjusted as more nodes join the network, thus increasing the total computational power of the network? Describe the process in detail.

26.17 Why is Byzantine consensus a poor consensus mechanism in a public blockchain?

26.18 Explain how off-chain transaction processing can enhance throughput. What are the trade-offs for this benefit?

26.19 Choose an enterprise of personal interest to you and explain how blockchain technology could be employed usefully in that business.

Tools

One can download blockchain software to create a full node for public blockchains such as Bitcoin (bitcoin.org) and Ethereum (www.ethereum.org) and begin mining, though the economic return for the investment of power may be questionable. Tools exist also to join mining pools. Browsing tools exist to view the contents of public blockchains. For some blockchains, notably Ethereum, it is possible to install a private copy of the blockchain software managing a private blockchain as an educational tool. Ethereum also offers a public test network where smart contracts can be debugged without the expense of gas on the real network.

Hyperledger (www.hyperledger.org) which is supported by a large consortium of companies, provides a wide variety of open source blockchain platforms and tools. Corda (www.corda.net) and BigchainDB (www.bigchaindb.com) are two other blockchain platforms, with BigchainDB having a specific focus on blockchain databases.

Blockchain based systems for supporting academic certificates and medical records, such as Blockcert and Medrec (both from MIT), and several other applications are available online. The set of tools for blockchain are evolving rapidly. Due to the rapid rate of change and development, as of late 2018 we are unable to identify a best set of tools, beyond the few mentioned above, that we can recommend. We recommend you perform a web search for the latest tools.

Further Reading

The newness of blockchain technology and applications means that, unlike the more established technical topics elsewhere in this text, there are fewer references in the academic literature and fewer textbooks. Many of the key papers are published only on the website of a particular blockchain. The URLs for those references are likely to change often. Thus, web searches for key topics are a highly important source for further reading. Here, we cite some classic references as well as URLs current as of the publication date.

Telegram: @uni_k

www.konkur.in

The original Bitcoin paper [Nakamoto (2008)] is authored under a pseudonym, with the identity of the author or authors still the subject of speculation. The original Ethereum paper [Buterin (2013)] has been superseded by newer Ethereum white papers (see ethereum.org), but the original work by Ethereum's creator, Vitalik Buterin, remains interesting reading. Solidity, the primary programming language for Ethereum smart contracts, is discussed in solidity.readthedocs.io. The ERC-20 standard is described in [Vogelsteller and Buterin (2013)] and the proposed (as of the publication date of this text) Casper upgrade to the performance of Ethereum's consensus mechanism appears in [Buterin and Griffith (2017)]. Another approach to using proof-of-stake is used by the Cardano blockchain (www.cardano.org).

Many of the theoretical results that make blockchain possible were first developed in the 20th century. The concepts behind cryptographic hash functions and public-key encryption were introduced in [Diffie and Hellman (1976)] and [Rivest et al. (1978)]. A good reference for cryptography is [Katz and Lindell (2014)]. [Narayanan et al. (2016)] is a good reference for the basics of cryptocurrency, though its focus is mainly on Bitcoin. There is a large body of literature on Byzantine consensus. Early papers that laid the foundation for this work include [Pease et al. (1980)] and [Lamport et al. (1982)]. Practical Byzantine fault tolerance ([Castro and Liskov (1999)]) serves as the basis for much of the current blockchain Byzantine consensus algorithms. [Mazières (2016)] describes in detail a consensus protocol specifically designed to allow for open, rather than permissioned, membership in the consensus group. References pertaining to Merkle trees appears in Chapter 23. Patricia trees were introduced in [Morrison (1968)].

A benchmarking framework for permissioned blockchains appears in [Dinh et al. (2017)]. A detailed comparison of blockchain systems appears in [Dinh et al. (2018)]. ForkBase, a storage system designed for improved blockchain performance, is discussed in [Wang et al. (2018)].

The Lightning network(lightning.network) aims to accelerate Bitcoin transactions and provide some degree of cross-chain transactions. Ripple (ripple.com) provides a network for international fiat currency exchange using the XRP token. Loopring (loopring.org) is a cryptocurrency exchange platform that allows users to retain control of their currency without having to surrender control to the exchange.

Many of the blockchains discussed in the chapter have their best descriptions on their respective web sites. These include Corda (docs.corda.net), Iota (iota.org), and Hyperledger (www.hyperledger.org). Many financial firms are creating their own blockchains, and some of those are publicly available, including J.P. Morgan's Quorum (www.jpmorgan.com/global/Quorum).

Bibliography

[Buterin (2013)] V. Buterin, "Ethereum: The Ultimate Smart Contract and Decentralized Application Platform", Technical report (2013).

Telegram: @uni_k

www.konkur.in

[Buterin and Griffith (2017)] V. Buterin and V. Griffith, "Casper the Friendly Finality Gadget", Technical report (2017).

[Castro and Liskov (1999)] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", In *Symp. on Operating Systems Design and Implementation (OSDI)*, USENIX (1999).

[Diffie and Hellman (1976)] W. Diffie and M. E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Volume 22, Number 6 (1976).

[Dinh et al. (2017)] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2017), pages 1085–1100.

[Dinh et al. (2018)] T. T. A. Dinh, R. Liu, M. H. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling Blockchain: A Data Processing View of Blockchain Systems", volume 30 (2018), pages 1366–1385.

[Katz and Lindell (2014)] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 3rd edition, Chapman and Hall/CRC (2014).

[Lamport et al. (1982)] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 3 (1982), pages 382–401.

[Mazières (2016)] D. Mazières, "The Stellar Consensus Protocol", Technical report (2016).

[Morrison (1968)] D. Morrison, "Practical Algorithm To Retrieve Information Coded in Alphanumeric", *Journal of the ACM*, Volume 15, Number 4 (1968), pages 514–534.

[Nakamoto (2008)] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", Technical report, Bitcoin.org (2008).

[Narayanan et al. (2016)] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies*, Princeton University Press (2016).

[Pease et al. (1980)] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, Volume 27, Number 2 (1980), pages 228–234.

[Rivest et al. (1978)] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, Volume 21, Number 2 (1978), pages 120–126.

[Vogelsteller and Buterin (2013)] F. Vogelsteller and V. Buterin, "ERC-20 Token Standard", Technical report (2013).

[Wang et al. (2018)] S. Wang, T. T. A. Dihn, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "ForkBase: An Efficient Storage Engine for Blockchain and Forkable Applications", In *Proc. of the International Conf. on Very Large Databases* (2018), pages 1085–1100.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

Telegram: @uni_k

www.konkur.in



PART 10

APPENDIX A

Appendix A presents the full details of the university database that we have used as our running example, including an E-R diagram, SQL DDL, and sample data that we have used throughout the book. (The DDL and sample data are also available on the web site of the book, db-book.com, for use in laboratory exercises.)

Telegram: @uni_k

www.konkur.in

Telegram: @uni_k

www.konkur.in

APPENDIX A



Detailed University Schema

In this appendix, we present the full details of our running-example university database. In Section A.1 we present the full schema as used in the text and the E-R diagram that corresponds to that schema. In Section A.2 we present a relatively complete SQL data definition for our running university example. Besides listing a datatype for each attribute, we include a substantial number of constraints. Finally, in Section A.3, we present sample data that correspond to our schema. SQL scripts to create all the relations in the schema, and to populate them with sample data, are available on the web site of the book, db-book.com.

A.1 Full Schema

The full schema of the university database that is used in the text follows. The corresponding schema diagram, and the one used throughout the text, is shown in Figure A.1.

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

Telegram: @uni_k

www.konkur.in

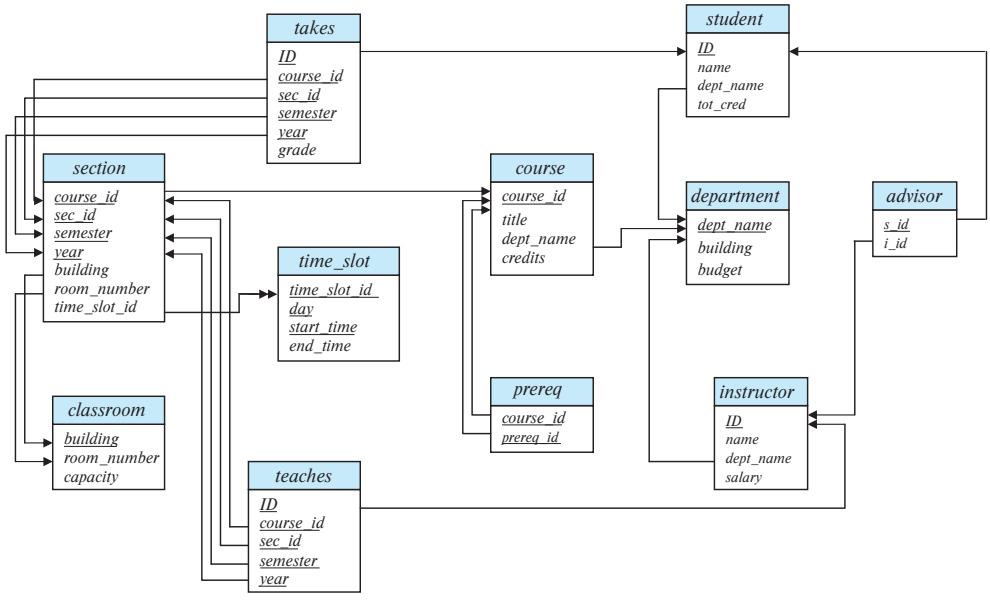


Figure A.1 Schema diagram for the university database.

A.2 DDL

In this section, we present a relatively complete SQL data definition for our example. Besides listing a datatype for each attribute, we include a substantial number of constraints.

```
create table classroom
    (building       varchar (15),
     room_number    varchar (7),
     capacity       numeric (4,0),
     primary key (building, room_number));

create table department
    (dept_name      varchar (20),
     building       varchar (15),
     budget         numeric (12,2) check (budget > 0),
     primary key (dept_name));
```

Telegram: @uni_k

www.konkur.in


```
create table course
    (course_id      varchar (7),
     title          varchar (50),
     dept_name      varchar (20),
     credits        numeric (2,0) check (credits > 0),
     primary key (course_id),
     foreign key (dept_name) references department
                    on delete set null);

create table instructor
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     salary         numeric (8,2) check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department
                    on delete set null);

create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6) check (semester in
                            ('Fall', 'Winter', 'Spring', 'Summer')),
     year           numeric (4,0) check (year > 1701 and year < 2100),
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course
                    on delete cascade,
     foreign key (building, room_number) references classroom
                    on delete set null);
```

In the preceding DDL, we add the **on delete cascade** specification to a foreign key constraint if the existence of the tuple depends on the referenced tuple. For example, we add the **on delete cascade** specification to the foreign key constraint from *section* (which was generated from weak entity *section*), to *course* (which was its identifying relationship). In other foreign key constraints we either specify **on delete set null**, which allows deletion of a referenced tuple by setting the referencing value to null, or we do not add any specification, which prevents the deletion of any referenced tuple. For example, if a department is deleted, we would not wish to delete associated instructors;

Telegram: @uni_k

www.konkur.in

the foreign key constraint from *instructor* to *department* instead sets the *dept_name* attribute to null. On the other hand, the foreign key constraint for the *prereq* relation, shown later, prevents the deletion of a course that is required as a prerequisite for another course. For the *advisor* relation, shown later, we allow *i_ID* to be set to null if an instructor is deleted but delete an *advisor* tuple if the referenced student is deleted.

```
create table teaches
    (ID             varchar (5),
     course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section
                    on delete cascade,
     foreign key (ID) references instructor
                    on delete cascade);

create table student
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     tot_cred       numeric (3,0) check (tot_cred >= 0),
     primary key (ID),
     foreign key (dept_name) references department
                    on delete set null);

create table takes
    (ID             varchar (5),
     course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     grade          varchar (2),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section
                    on delete cascade,
     foreign key (ID) references student
                    on delete cascade);
```

Telegram: @uni_k

www.konkur.in


```
create table advisor
    (s_ID          varchar (5),
     i_ID          varchar (5),
     primary key (s_ID),
     foreign key (i_ID) references instructor (ID)
                   on delete set null,
     foreign key (s_ID) references student (ID)
                   on delete cascade);

create table prereq
    (course_id     varchar(8),
     prereq_id     varchar(8),
     primary key (course_id, prereq_id),
     foreign key (course_id) references course
                   on delete cascade,
     foreign key (prereq_id) references course);
```

The following **create table** statement for the table *time_slot* can be run on most database systems, but it does not work on Oracle (at least as of Oracle version 11), since Oracle does not support the SQL standard type **time**.

```
create table timeslot
   (time_slot_id    varchar (4),
    day             varchar (1) check (day in ('M', 'T', 'W', 'R', 'F', 'S', 'U')),
    start_time      time,
    end_time        time,
    primary key (time_slot_id, day, start_time));
```

The syntax for specifying time in SQL is illustrated by these examples: '08:30', '13:55', and '5:30 PM'. Since Oracle does not support the **time** type, for Oracle we use the following schema instead:

```
create table timeslot
   (time_slot_id    varchar (4),
    day             varchar (1),
    start_hr        numeric (2) check (start_hr >= 0 and end_hr < 24),
    start_min       numeric (2) check (start_min >= 0 and start_min < 60),
    end_hr          numeric (2) check (end_hr >= 0 and end_hr < 24),
    end_min         numeric (2) check (end_min >= 0 and end_min < 60),
    primary key (time_slot_id, day, start_hr, start_min));
```

The difference is that *start_time* has been replaced by two attributes *start_hr* and *start_min*, and similarly *end_time* has been replaced by attributes *end hr* and *end min*.

Telegram: @uni_k

www.konkur.in

These attributes also have constraints that ensure that only numbers representing valid time values appear in those attributes. This version of the schema for *time_slot* works on all databases, including Oracle. Note that although Oracle supports the **datetime** datatype, **datetime** includes a specific day, month, and year as well as a time, and is not appropriate here since we want only a time. There are two alternatives to splitting the time attributes into an hour and a minute component, but neither is desirable. The first alternative is to use a **varchar** type, but that makes it hard to enforce validity constraints on the string as well as to perform comparison on time. The second alternative is to encode time as an integer representing a number of minutes (or seconds) from midnight, but this alternative requires extra code with each query to convert values between the standard time representation and the integer encoding. We therefore choose the two-part solution.

A.3 Sample Data

In this section we provide sample data for each of the relations defined in the previous section.

Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure A.2 The *classroom* relation.

Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure A.3 The *department* relation.

Telegram: @uni_k

www.konkur.in

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure A.4 The *course* relation.

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure A.5 The *instructor* relation.

Telegram: @uni_k

www.konkur.in

BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

Figure A.6 The *section* relation.

10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure A.7 The *teaches* relation.

Telegram: @uni_k

www.konkur.in

00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure A.8 The *student* relation.

Telegram: @uni_k

www.konkur.in

00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

Figure A.9 The *takes* relation.

00128	45565
12345	10101
23121	76543
44553	22222
45678	22222
76543	45565
76653	98345
98765	98345
98988	76766

Figure A.10 The *advisor* relation.

Telegram: @uni_k

www.konkur.in

A	M	8:00	8:50
A	W	8:00	8:50
A	F	8:00	8:50
B	M	9:00	9:50
B	W	9:00	9:50
B	F	9:00	9:50
C	M	11:00	11:50
C	W	11:00	11:50
C	F	11:00	11:50
D	M	13:00	13:50
D	W	13:00	13:50
D	F	13:00	13:50
E	T	10:30	11:45
E	R	10:30	11:45
F	T	14:30	15:45
F	R	14:30	15:45
G	M	16:00	16:50
G	W	16:00	16:50
G	F	16:00	16:50
H	W	10:00	12:30

Figure A.11 The *time_slot* relation.

BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure A.12 The *prereq* relation.

Telegram: @uni_k

www.konkur.in

A	M	8	0	8	50
A	W	8	0	8	50
A	F	8	0	8	50
B	M	9	0	9	50
B	W	9	0	9	50
B	F	9	0	9	50
C	M	11	0	11	50
C	W	11	0	11	50
C	F	11	0	11	50
D	M	13	0	13	50
D	W	13	0	13	50
D	F	13	0	13	50
E	T	10	30	11	45
E	R	10	30	11	45
F	T	14	30	15	45
F	R	14	30	15	45
G	M	16	0	16	50
G	W	16	0	16	50
G	F	16	0	16	50
H	W	10	0	12	30

Figure A.13 The *time_slot* relation with start and end times separated into hour and minute.

Telegram: @uni_k

www.konkur.in

Index

aborted transactions, 805–807, 819–820
abstraction, 2, 9–12, 15
acceptors, 1148, 1152
accessing data. ***See also*** **security**
from application programs, 16–17
concurrent-access anomalies, 7
difficulties in, 6
indices for, 19
recovery systems and, 910–912
types of access, 15
access paths, 695
access time
indices and, 624, 627–628
query processing and, 692
storage and, 561, 566, 567, 578
access types, 624
account nonces, 1271
ACID properties. ***See*** **atomicity; consistency; durability;** isolation
Active Server Page (ASP), 405
active transactions, 806
ActiveX DataObjects (ADO), 1239
adaptive lock granularity, 969–970
add constraint, 146
ADO (ActiveX DataObjects), 1239
ADO.NET, 184, 1239
Advanced Encryption Standard (AES), 448, 449
advanced SQL, 183–231
accessing from programming languages, 183–198
aggregate features, 219–231
embedded, 197–198
functions and procedures, 198–206
JDBC and, 184–193
ODBC and, 194–197
Python and, 193–194
triggers and, 206–213
advertisement data, 469
AES (Advanced Encryption Standard), 448, 449
after triggers, 210
aggregate functions, 91–96
basic, 91–92
with Boolean values, 96
defined, 91
with grouping, 92–95
having clause, 95–96
with null values, 96
aggregation
defined, 277
entity-relationship (E-R) model and, 276–277
intraoperation parallelism and, 1049
on multidimensional data, 527–532
partial, 1049
pivoting and, 226–227, 530
query optimization
query processing and, 723
ranking and, 219–223
representation of, 279
rollup and cube, 227–231
skew and, 1049–1050
of transactions, 1278
view maintenance and, 781–782
windowing and, 223–226
aggregation operation, 57
aggregation switch, 977
airlines, database applications for, 3
Ajax, 423–426, 1015
algebraic operations. ***See*** **relational algebra**
aliases, 81, 336, 1242
all construct, 100
alter table, 71, 146
alter trigger, 210
alter type, 159
Amdahl's law, 974
American National Standards Institute (ANSI), 65, 1237
analysis pass, 944
analytics. ***See*** **data analytics**
and connective, 74
and operation, 89–90
anonymity, 1252, 1253, 1258, 1259
ANSI (American National Standards Institute), 65, 1237
anticipatory standards, 1237

Mc Graw Hill Education
5011111
Telegram: @uni_k

www.konkur.in

anti-semijoin operation, 776–777
Apache
AsterixDB, 668
Cassandra, 477, 489, 668, 1024, 1028
Flink project, 504, 508
Giraph system, 511
HBase, 477, 480, 489, 668, 971, 1024, 1028–1031
Hive, 494, 495, 500
Jakarta Project, 416
Kafka system, 506, 507, 1072–1073, 1075, 1137
Spark, 495–500, 508, 511, 1061
Storm stream-processing system, 506–508
Tez, 495
APIs. *See* application program interfaces
application design, 403–453
application architectures and, 429–434
authentication and, 441–443
business logic and, 23, 404, 411–412, 430, 431, 445
client-server architecture and, 404
client-side code and web services, 421–429
common gateway interface standard and, 409
cookies and, 410–415, 411n2
data access layer and, 430–434
disconnected operation and, 427–428
encryption and, 447–453
HTML and, 404, 406–408, 426
HTTP and, 405–413
JavaScript and, 404–405, 421–426
Java Server Pages and, 405, 417–418
mobile application platforms, 428–429
performance and, 434–437
security and, 437–446
servlets and, 411–421
standardization and, 1237–1240
testing, 1234–1235
tuning and (*see* performance tuning)
URLs and, 405–406
user interfaces and, 403–405
web and, 405–411
application migration, 1035–1036
application program interfaces (APIs)
ADO, 1239
ADO.NET, 184, 1239
application design and, 411, 413, 416
C++, 1239
database access from, 16–17
Java (*see* Java)
LDAP, 1243
map displays and, 393
MongoDB, 477–479, 482, 489, 668, 1024, 1028
Open Database Connectivity (*see* ODBC)
Python (*see* Python)
Spark, 495–500, 508, 511, 1061
standards for, 1238–1240
system architectures and, 962
Tez, 495
web services and, 427
application programmers, 24
application servers, 23, 416
architectures, 961–995
business logic and, 23
centralized databases, 962–963
client-server systems, 23, 971
cloud-based, 990–995, 1026, 1027
database storage, 587–588
data-server systems, 963–964, 968–970
data warehousing, 522–523
destination-driven, 522
distributed databases, 22, 986–989, 1098–1100
hierarchical, 979, 980, 986
hypercube, 976–977
lambda, 504, 1071
mesh, 976
microservices, 994
multiuser systems, 962
Non-Uniform Memory Access, 981, 1063
overview, 961–962
parallel databases, 22, 970–986
platform-as-a-service model, 992–993
recovery systems, 932
server system, 962–970, 977–978
shared disk, 979, 980, 984–985
shared memory, 21–22, 979–984, 1061–1064
shared nothing, 979, 980, 985–986, 1040–1041, 1061–1063
single-user systems, 962
software-as-a-service model, 993
source-driven, 522
storage area network, 562
three-tier, 23
transaction-server systems, 963–968
two-phase commit protocol, 989, 1276
two-tier, 23
wide-area networks, 989
archival data, 561
archival dump, 931
ARIES
analysis pass and, 944
compensation log records and, 942, 945
data structures and, 942–944
dirty page table and, 941–947
fine-grained locking and, 947
fuzzy checkpoints and, 941
log sequence number and, 941–946
nested top actions and, 946
optimization and, 947
physiological redo and, 941
recovery algorithm, 944–946,

Telegram: @uni_k

www.konkur.in

redo pass and, 944–945
rollback and, 945–946
savepoints and, 947
undo pass and, 944–946
arity, 54
Armstrong's axioms, 321
array databases, 367
array types, 366, 367, 378
asc expression, 84
as clause, 79, 81
as of period for, 157
ASP (Active Server Page), 405
ASP.NET, 417
assertions, 152–153
asset management, 1277
assignment operation, 55–56, 201
associations
data mining and, 541
entity sets (*see* entity sets)
relation schema for, 42
relationship sets (*see* relationship sets)
rules for, 546–547
associative property, 749–750
AsterixDB, 668
asymmetric fragment-and-replicate joins, 1046, 1062
asymmetric-key encryption, 448
asynchronous replication, 1122, 1135–1138
asynchronous view maintenance, 1138–1140
at-least-once semantics, 1074
at-most-once semantics, 1074
atomic commit, 1029
atomic domains, 40, 342–343
atomic instructions, 966–967
atomicity
cascadeless schedules and, 820–821
commit protocols and, 1100–1110
defined, 20, 800
in file-processing systems, 6–7
isolation and, 819–821
log records and, 913–919
recoverable schedules and, 819–820
recovery systems and, 803, 912–922
storage structure and, 804–805
of transactions, 20–21, 144, 481, 800–807, 819–821
attribute inheritance, 274–275
attributes
atomic domains and, 342–343
bitmap indices and, 670–672
classifiers and, 541–543, 545
closure of attribute sets, 322–324
complex, 249–252, 265–267
composite, 250, 252
decomposition and, 305–313, 330–335, 339–340
derived, 251, 252
descriptive, 248
design issues and, 281–282
discriminator, 259
domain of, 39–40, 249
entity-relationship diagrams and, 265–267
entity-relationship (E-R) model and, 245, 248–252, 274–275, 281–282, 342–343
entity sets and, 245, 265–267, 281–282
extraneous, 325
histograms and, 758–760
multiple-key access and, 661–663
multivalued, 251, 252, 342
naming of, 345–346
null values and, 251–252
partitioning, 479
primary keys and, 310n4
prime, 356
in relational model, 39
relationship sets and, 248
search key and, 624
simple, 250, 265
single-valued, 251
Unified Modeling Language and, 289
uniquifiers, 649–650
value set of, 249
attribute-value skew,
audit trails, 445–446
augmentation rule, 321
authentication
application-level, 441–443
challenge-response system and, 451
digital certificates and, 451–453
digital signatures and, 451
encryption and, 450–453
single sign-on system and, 442–443
smart cards and, 451, 451n9
two-factor, 441–442
web sessions and, 410
authorization
administrators and, 166
application-level, 443–445
database design and, 291
end-user information and, 443
granting privileges, 25, 166–167, 170–171
lack of fine-grained, 443–445
permissioned blockchains and, 1253
revoking privileges, 166–167, 171–173
roles and, 167–169
row-level, 173
on schema, 170
Security Assertion Markup Language and, 442–443
SQL DDL and, 66
sql security invoker, 170
storage manager and, 19
transfer of privileges, 170–171
types of, 14, 165
updates and, 14, 170, 171
on views, 169–170
authorization graph, 171
automatic commit, 144, 144n6, 822
autonomous smart contracts, 1272–1273
autonomy, 988
availability
CAP theorem and, 1134
distributed databases and,

Telegram: @uni_k

www.konkur.in

high availability, 907, 931–933, 987, 1121
network partitions and, 481, 989
robustness and, 1121
trading off consistency for, 1134–1135
average latency time, 566
average response time, 809
average seek time, 566
avg expression, 91–96, 723, 781–782
Avro, 490, 499
axioms, 321
Azure Stream Analytics, 505

backpropagation algorithm, 545
backup. ***See also*** **recovery systems**
application design and, 450
remote systems for, 909, 931–935
replica systems, 212–213
transactions and, 805
backup coordinators, 1146–1147
balanced range-partitioning vector, 1008–1009
balanced trees, 634
banking
analytics for, 520–521
database applications for, 3, 4, 7, 144
BASE properties, 1135
base query, 217
batch scaleup, 973
batch update, 1221
Bayesian classifiers, 543–544
Bayes' theorem, 543
BCNF. ***See*** **Boyce–Codd normal form**
BCNF decomposition algorithm, 331–333, 336
before triggers, 210
begin atomic…end, 145, 201, 208, 209, 211
begin transaction operation, 799
benchmarks. ***See*** **performance benchmarks**
bestplan array, 768–770
biased protocol, 1124
BI (business intelligence), 521
big-bang approach, 1236
BigchainDB, 1269
Big Data, 467–511
algebraic operations and, 494–500
comparison with traditional databases, 468
defined, 467
distributed file systems for, 472–475, 489
graph databases and, 508–511
key-value systems and, 471, 473, 476–480
MapReduce paradigm and, 481, 483–494
motivations for, 467–472
parallel and distributed databases for, 473, 480–481
query processing and, 470–472
replication and consistency in, 481–482
sharding and, 473, 475–476
sources and uses of, 468–470
storage systems, 472–482, 668
streaming data, 468, 500–508
Bigtable, 477, 479–480, 668, 1024–1025, 1028–1030
binary operations, 48
binary relationship sets, 249, 283–285
Bing Maps, 393
Bitcoin
anonymity and, 1253, 1258
data mining and, 1265
forking and, 1258
growth and development of, 1251–1253
language and, 1269–1270
processing speed, 1274
as public blockchain, 1253, 1255, 1263
transactions and, 1261–1263, 1268–1271
bit-level striping, 571–572
bitmap indices
attributes and, 670–672
B+-trees and, 1185–1186
efficient implementation of, 1184–1185
existence, 1184
intersection and, 671, 1183
processing speed and, 662, 663
scans of, 698–699
sequential records and, 1182
structure of, 1182–1184
usefulness of, 671–672
bit rot, 575
blind writes, 868
B-link trees, 886
blobs, 156, 193, 594, 652
blockchain databases, 1251–1279
anonymity and, 1252, 1253, 1258, 1259
applications for use, 1251–1252, 1276–1279
concurrency control and, 1262–1263
consensus mechanisms for, 1254, 1256–1257, 1263–1267
cryptocurrencies and, 1251–1253, 1257
cryptographic hash functions and, 1253, 1259–1263, 1265
data mining and, 1256, 1258, 1264–1266
decentralization and, 1251, 1252, 1259, 1270
digital ledgers and, 1251, 1252
digital signatures and, 1257, 1261
encryption and, 1260–1261
external input and, 1271–1272
fault-tolerance of, 1276
forking and, 1257, 1258, 1263
genesis blocks and, 1254–1255
irrefutability and, 1257, 1259
languages and, 1258, 1269–1271
lookup in, 1268
management of data in,

Telegram: @uni_k

www.konkur.in

orphaned blocks and, 1257, 1263
performance enhancement of, 1274–1276
permissioned, 1253–1254, 1256–1257, 1263, 1266, 1274
properties and components of, 1254–1259, 1274
public, 1253, 1255, 1257–1259, 1263, 1264
query processing and, 1254, 1275–1276
scalability of, 1276
smart contracts and, 1258, 1269–1273
state-based, 1269, 1271
tamper resistant nature of, 1253–1255, 1259, 1260
transactions and, 1261–1263, 1268–1271, 1273
block identifiers, 474–475, 1020
blocking edges, 728
blocking factor, 725
blocking operations, 728, 728n7
blocking problem, 1104–1106
block-interleaved parity organization, 573
block-level striping, 572
block nested-loop join, 705–707
block-oriented interface, 560
blocks
buffer, 910–912
dirty, 928–929
disk, 566–567, 577–580
evicted, 605
file organization and, 588
genesis, 1254–1255
orphaned, 1257, 1263
overflow, 598
physical, 910
pinned, 605
Bloom filters, 667, 1083, 1175–1176, 1181
Boolean operations, 89, 96, 103, 188, 201, 1242. ***See*** also ***specific operations***
bottlenecks
application design and, 437
I/O parallelism and, 1007
performance tuning and, 1211–1213, 1215, 1227
single lock-manager and, 1111
system architectures and, 981, 985
bottom-up B$^+$-tree construction, 654–655
bottom-up design, 273
bounding boxes, 674–675, 1187
Boyce–Codd normal form (BCNF)
comparison with third normal form, 318–319
decomposition algorithm and, 331–333, 336
defined, 313–315
dependency preservation and, 315–316
relational database design and, 313–316
testing for, 330–331
broadcasting, 1055–1056
broadcast join, 1046
BSP (bulk synchronous processing), 510–511
B-trees, comparison with B$^+$-trees, 655–656
B$^+$-trees, 634–658
balanced, 634
bitmap indices and, 1185–1186
bottom-up construction of, 654–655
bulk loading of, 653–655
comparison with B-trees, 655–656
deletion and, 641, 645–649
extensions and variations of, 650–658
fanout and, 635
on flash storage, 656–657
indexing strings and, 653
insertion and, 641–645, 647, 649
internal nodes and, 635
leaf nodes of, 635–656, 665–669, 673, 674
in main memory, 657–658
nonleaf nodes of, 635–636, 642, 645–656,
nonunique search keys and, 649–650
organization of, 595, 650–652, 697, 697n4
parallel key-value stores and, 1028
performance and, 634, 665–666
queries on, 637–641, 690
record relocation and, 652–653
secondary indices and, 652–653
spatial data and, 672–673
structure of, 634–637
temporal data and, 676
tuning of, 1215
updates on, 641–649
buckets, 659–661, 1194–1195
buffer blocks, 910–912
buffer manager, 19, 604–607
buffers
database buffering, 927–929
defined, 604
disk blocks and, 578, 910
double buffering, 725
force/no-force policy and, 927
force output and, 912
log-record, 926–927
management of, 926–930
operating system role in, 929–930
output of blocks and, 606–607
recovery systems and, 926–930
reordering of writes and recovery, 609–610
replacement strategies, 605, 607–609
shared and exclusive locks on, 605–606
steal/no-steal policy and, 927
storage and, 604–610
transaction servers and, 965
write-ahead logging rule and, 926–929
buffer trees, 668–670

Telegram: @uni_k

www.konkur.in

application design and, 440, 1234, 1236
debugging, 199n4
failure classification and, 908
build input, 713
bulk export utility, 1222
bulk insert utility, 1222
bulk loads, 653–655, 1221–1223
bulk synchronous processing (BSP), 510–511
bully algorithm, 1148
business intelligence (BI), 521
business logic, 23, 198, 404, 411–412, 430–431, 445
business-logic layer, 430, 431
business rules, 431
bus system, 975–976
Byzantine consensus, 1254, 1256, 1266–1267, 1276
Byzantine failure, 1266–1267

C
advanced SQL and, 183, 197, 199, 205
application design and, 16
ODBC and, 195–196
struct declarations used by, 11n1
Unified Modeling Language and, 289
C++
advanced SQL and, 197, 199, 205, 206
application design and, 16, 417
object-oriented programming and, 377
standards for, 1239
struct declarations used by, 11n1
Unified Modeling Language and, 289
cache-conscious algorithms, 732–733
cache line, 732, 983
cache memory, 559
cache misses, 982
caching
application design and, 435–437
coherency and, 969, 983–984
column-oriented storage and, 612
data servers and, 968–970
locks and, 969
query plans and, 774, 965
replication and, 1014n4
shared-memory architecture and, 982–984
CAD (computer-aided design), 390–391, 968
callable statements, 190–191
call back, 969
Call Level Interface (CLI) standards, 197, 1238–1239
call statement, 201
candidate keys, 44
canonical cover, 324–328
CAP theorem, 1134
Cartesian-product operation, 50–52
Cartesian products
equivalence and, 748, 749, 755
join expressions and, 135
query optimization and, 748, 749, 755, 763–764, 775
SQL and, 76–79, 81, 127n1, 230
cascadeless schedules, 820–821
cascades, 150, 172, 210
cascading rollback, 820–821, 841–842
cascading stylesheet (CSS) standard, 408
case construct, 112–113
Cassandra, 477, 489, 668, 1024, 1028
cast, 155, 159
catalogs
application design and, 1239
indices and (*see* indices)
query optimization and, 758–760, 762, 764
SQL and, 162–163, 192, 196–197
system, 602–604, 1009
centralized databases, 962–963
centralized deadlock detection, 1114
centroid, 548, 548n3
CEP (complex event processing) systems, 504
CGI (common gateway interface) standard, 409
chain replication protocol, 1127–1128
challenge-response system, 451
change isolation level, 822
change relation, 211
char, 67
check clause
assertions and, 152–153
integrity constraints and, 147–149, 152–153
user-defined types and, 159
check constraints, 151, 170, 315, 800
checkpoint log records, 943
checkpoint process, 965
checkpoints
fuzzy, 922, 930, 941
recovery systems and, 920–922, 930
transaction servers and, 965
checksums, 565
chicken-little approach, 1236
Chubby, 1150
circular arcs, 388
classifiers
attributes and, 541–543, 545
Bayesian, 543–544
data mining and, 541–546
decision-tree, 542
neural-net, 545–546
prediction and, 541–543, 545–546
Support Vector Machine, 544–545
training instances and, 541
CLI (Call Level Interface) standards, 197, 1238–1239
click-through, 469
client-server systems
application design and, 404, 1221, 1239
recovery systems and, 931
system architecture and, 23,

Telegram: @uni_k

www.konkur.in

client-side scripting languages, 421–429
clobs, 156, 193, 594, 652
closed addressing, 659, 1194
closed hashing, 659, 1194
closed polygons, 388n3
closed time intervals, 675
closure of a set, 312, 320–324
cloud-based data storage, 28, 563, 992–993
cloud computing
architecture for, 990–995, 1026, 1027
benefits and limitations of, 995
service models, 991–995
storage systems and, 28, 563
CLR (Common Language Runtime), 206
CLRs (compensation log records), 922, 942, 945
clustering indices, 625, 632–633, 695, 697–698
cluster key, 600–601
cluster membership, 1158
clusters
data mining and, 541, 548–549
hierarchical, 548
key-value storage systems and, 477
multitable, 595, 598–601
system architecture and, 978
coalesce function, 114, 155, 230–231
coalescing nodes, 641, 886
coarse-grained parallelism, 963, 970
Codd, Edgar, 26
code breaking. *See* encryption
collision resistant hash functions, 1259–1260
colocation of data, 1068–1069
column family, 1025
column-oriented storage, 525–526, 588, 611–617, 734, 1182
column stores, 612, 615, 1025, 1224
combinatorics, 811
combine function, 490
comma-separated values, 1222
commit dependency, 847
commit protocols, 1100–1110
committed transactions
defined, 806
durability and, 933–934
log records and, 917
observable external writes and, 807
partially committed, 806
scheduling and, 810, 819–820
updates and, 874
commit time, 933–934
commit wait, 1130–1131
commit work, 143–145
common gateway interface (CGI) standard, 409
Common Language Runtime (CLR), 206
common subexpression elimination, 785
commutative property, 747–750
commute, 1143
compare-and-swap, 966
compatibility function, 836
compatible relations, 54
compensating operation, 892
compensating transactions, 805
compensation log records (CLRs), 922, 942, 945
complete axioms, 321
completeness constraint, 275
complex attributes, 249–252, 265–267
complex data types, 365–394
object orientation, 376–382
semi-structured, 365–376
spatial, 387–394
textual, 382–387
user-defined, 158
complex event processing (CEP) systems, 504
composite attributes, 250, 252
composite indices, 700
composite price/performance metric, 1234
composite query per hour metric, 1234
compression
column-oriented storage and, 611, 612
data warehousing and, 526
of disk block data, 615n8
prefix, 653
workload compression, 1217
computer-aided design (CAD), 390–391, 968
conceptual-design phase, 17–18, 242
concurrency control, 835–894
access anomalies and, 7
blind writes and, 868
blockchain databases and, 1262–1263
commit protocols and, 1105
consistency and, 880–885
deadlock handling and, 849–853
deletion and, 857–858
distributed databases and, 990, 1105, 1111–1120
extended protocols, 1129–1133
false cycles and, 1114–1115
in federated databases, 1132–1133
indices and, 884–887
insertion and, 857, 858
isolation and, 803–804, 807–812, 823
leases and, 1115–1116
locking protocols and, 835–848 (*see also* locks)
logical undo operations and, 940–941
long-duration transactions and, 890–891
in main-memory databases, 887–890
multiple granularity and, 853–857
multiversion schemes and, 869–872, 1129–1131
with operations, 891–894
optimistic, 869
parallel databases and, 990
parallel key-value stores and,

Telegram: @uni_k

www.konkur.in

phantom phenomenon and, 827, 858–861, 877–879, 877n5, 885, 887
predicate reads and, 858–861
real-time transaction systems and, 894
recovery systems and, 916
replication and, 1123–1125
rollback and, 841–844, 849–850, 853, 868–871
serializability and, 836, 840–843, 846–848, 856, 861–871, 875–887
snapshot isolation and, 872–879, 882, 916, 1131–1132
timestamp-based protocols and, 861–866, 882
trends in, 808
user interactions and, 881–883
validation and, 866–869, 882, 916
concurrency-control manager, 21
concurrency-control schemes, 809
concurrent transactions, 1224–1227
confidence, 540, 547
conflict equivalence, 815, 815n2
conflict serializability, 813–816
conformance levels, 196–197
conjunctive selection, 699–700, 747, 762
connection pooling, 436
consensus protocols
blockchain databases and, 1254, 1256–1257, 1263–1267
Byzantine, 1254, 1256, 1266–1267, 1276
distributed databases and, 1106–1107, 1150–1161, 1266, 1267
message-based, 1266
multiple consensus protocol, 1151
Paxos, 1152–1155, 1160–1161, 1267
proof-of-stake, 1256, 1266
proof-of-work, 1256, 1264–1266
Raft, 1148, 1155–1158, 1267
replication and, 1016
Zab, 1152
consistency
Big Data and, 481–482
CAP theorem and, 1134
concurrency control and, 880–885
cursor stability and, 881
deadlock and, 838–839
defined, 20
degree-two, 880–881
eventual, 1016, 1139
external, 1131
file system consistency check, 610
hashing and, 1013
logical operations and, 936–937
replication and, 1015–1016, 1121–1123, 1133–1146
requirement of, 802
trading off for availability, 1134–1135
of transactions, 20, 800, 802, 807–808, 821–823
user interactions and, 881–883
weak levels of, 880–883
constraints
check, 151, 170, 315
completeness, 275
consistency, 13–14
deadlines, 894
decomposition and, 336
dependency preservation and, 315–316
entity-relationship (E-R) model and, 253–256, 275–276
foreign key, 45–46
integrity (*see* integrity constraints)
keys and, 258
mapping cardinalities and, 253–256
not null, 69
primary key, 44
on specialization, 275–276
transactions and, 800
Unified Modeling Language and, 289
containers, 992–994
contains operation, 101
continuous queries, 503, 731
continuous-stream data, 731
conversations, 883
conversions, 155–156, 469, 843
cookies, 410–415, 411n2, 439–440
coordinators, 1099, 1104, 1106–1107, 1146–1150
Corda, 1269
cores, 962–963, 970, 976, 980–983
core switch, 977
correlated evaluation, 775
correlated subqueries, 101
correlation name, 81, 101
correlation variables, 81, 775
cost-based optimizers, 766
Couchbase, 1024
count function, 91–92, 94, 96, 723, 766, 781
count values, 220n11
covering indices, 663
crabbing protocol, 885–886
crashes. ***See also*** **recovery systems**
actions following, 923–925
algorithms for, 922–925
ARIES and, 941–947
checkpoints and, 920–922
failure classification and, 908
magnetic disks and, 565
storage and, 607, 609–610
transactions and, 800
crawling the web, 383
create assertion, 153
create cluster, 601
create distinct type, 160
create domain, 159–160
create function, 200, 203, 204, 215
create index, 164–165, 664
create or replace, 199n4
create procedure, 200, 205

Telegram: @uni_k

www.konkur.in

create role, 168
create schema, 163
create sequence construct, 161
create table
with data clause, 162
default values and, 156
extensions for, 162
integrity constraints and, 146–149
multitable clustering and, 601
object-based databases and, 378–380
shipping SQL statements to database and, 187
SQL schema definition and, 68–71
create table…as, 162
create table…like, 162
create temporary table, 214
create type, 158–160, 378–380
create unique index, 165, 664
create view, 138–143, 162, 169
credit bureaus, 521, 521n1
cross-chain transactions, 1273
cross join, 127n1
cross-site request forgery (XSRF), 439–440
cross-site scripting (XSS), 439–440
cross-tabulation, 226–227, 528–533
CRUD web interfaces, 419
cryptocurrencies, 1251–1253, 1257. *See also* Bitcoin
cryptographic hash functions, 1253, 1259–1263, 1265
CSS (cascading stylesheet) standard, 408
C-Store, 615
cube construct, 227–231, 536–538
current date, 154
cursor stability, 881
curve fitting, 546
cylinders, 565
Cypher query language, 509

DAGs (directed acyclic graphs), 499, 506–507, 1071–1072
DAOs (distributed autonomous organizations), 1272, 1272n7
Dart language, 428–429
data abstraction, 2, 9–12, 15
data access layer, 430–434
data analytics, 519–549. *See also* data mining
decision-support systems and, 519–522
defined, 4, 519
OLAP systems, 520, 527–540
overview, 519–521
predictive models in, 4–5
statistical analysis, 520, 527
warehousing and, 519–527
data-at-rest, 502
database administrators (DBAs), 24–25
database-as-a-service platform, 993
database design
alternatives in, 243–244, 285–291
applications and (*see* application design)
architecture of (*see* architectures)
authorization requirements and, 291
bottom-up, 273
buffers and, 604–610
client-server (*see* client-server systems)
complexity of, 241
computer-aided, 390–391
conceptual-design phase of, 17–18, 242
direct design process, 241
encryption and, 447–453
engines, 18–21
E-R model and (*see* entity-relationship model)
functional requirements of, 291
incompleteness in, 243–244
logical-design phase of, 18, 242
normalization in, 17
overview of process, phases of, 17–18, 241–243
physical-design phase of, 18, 242–243
redundancy in, 243
relational (*see* relational database design)
schema evolution and, 292
specification of functional requirements in, 17–18
top-down, 273
user requirements in, 17–18, 241–242, 274
workflow and, 291–292
database graph, 846–848
database instance, 41
database-management systems (DBMSs)
defined, 1
objectives of, 1, 24
organizational data processing prior to, 472
product-specific calls needed by, 186
databases
abstraction and, 2, 9–12, 15
administrators of, 24–25
applications for, 1–5
architecture (*see* architectures)
array, 367
blockchain (*see* blockchain databases)
buffering and, 927–929
centralized, 962–963
concurrency control and (*see* concurrency control)
defined, 1
design of (*see* database design)
document, 3
dumping and, 930–931
efficiency of, 1, 2, 5, 9
embedded, 198, 962
as file-processing systems, 5–8
force output and, 912
graph, 508–511
history of, 25–28
indexing and (*see* indices)
languages for, 13–17

Telegram: @uni_k

www.konkur.in

main memory (*see* main-memory databases)
maintenance for, 25
modification of, 108–114, 915–916
object-based (*see* object-based databases)
object-oriented, 9, 26, 377, 431, 1239–1240
object-relational, 377–381
parallel (*see* parallel databases)
purpose of, 5–8
query processor components of, 18, 20
recovery of (*see* recovery systems)
storage for (*see* storage)
transaction manager in, 18–21
university (*see* university databases)
user interaction with, 4–5, 24
databases administrator (DBA), 171
database schema. *See* schemas
database writer process, 965
data center fabric, 978
data centers, 970, 1014–1015
data cleansing, 523
data cubes, 529–530
data-definition language (DDL)
application programs and, 17
authorization and, 66
basic types supported by, 67–68
in consistency constraint specification, 13–14
defined, 13, 65
dumping and, 931
granting and revoking privileges and, 166
indices and, 67
integrity and, 66
interpreter, 20
output of, 14
schema definition and, 24, 66, 68–71
security and, 67
set of relations in, 66–67
SQL and, 14–15, 65–71
storage and, 67
data dictionary, 14, 19, 602–604
data distribution skew, 1008
Data Encryption Standard (DES), 448
data files, 19
data inconsistency, 6
data isolation. *See* isolation
data-item identifiers, 913
data items, 968
data lakes, 527, 1078
data-manipulation language (DML)
application programs and, 17
compiler, 20
declarative, 15
defined, 13, 15, 66
procedural, 15
SQL and, 16
storage manager and, 19
data mining, 540–549
association rules and, 546–547
blockchain databases and, 1256, 1258, 1264–1266
classifiers and, 541–546
clustering and, 541, 548–549
defined, 5, 540
descriptive patterns and, 541
growth of, 27
models for, 540
overview, 521
prediction and, 541
regression and, 546
rules for, 540
task types in, 541
text mining, 549
data models, 8–9. *See also specific models*
datanodes, 475, 1020
data parallelism, 1042, 1057
data partitioning, 989n5
data-server systems, 963–964, 968–970
DataSet type, 499
data storage and definition language, 13
data storage systems. *See* storage
data streams, 731
data striping, 571–572
data-transfer failures, 909
data-transfer rate, 566, 569
data types. *See* types
data virtualization, 1077
data visualization, 538–540
data warehousing, 519–527
architecture for, 522–523
column-oriented storage and, 525–526
components of, 522–524
database support for, 525–526
data integration vs., 1077–1078
data lakes and, 527, 1078
deduplication and, 523
defined, 519, 522
ETL tasks and, 520, 524
fact tables and, 524
householding and, 523
merger-purge operation and, 523
multidimensional data and, 524
overview, 519–520
schemas used for, 523–525
transformation and cleansing, 523
updates and, 523
datetime data type, 154, 531
DBAs (database administrators), 24–25
DBMSs. *See* database-management systems
DDL. *See* data-definition language
DDL interpreter, 20
deadlines, 894
deadlocks
consistency and, 838–839
detection of, 849, 851–852
distributed databases and, 1111–1115
handling of, 849–853
prevention of, 849–851
recovery and, 849, 851, 853
rollback and, 853
starvation and, 853

Telegram: @uni_k

www.konkur.in

wait-for graphs and, 851–852, 1113–1114
debugging, 199n4
decentralization, 1251, 1252, 1259, 1270
decision support, 521, 1231–1233
decision-support queries, 521, 971
decision-support systems, 519–522
decision-support tasks, 521
decision-tree classifiers, 542
declarative DMLs, 15
declarative queries, 47, 1030–1031
declare statement, 201–203
decode, 155–156
decomposition
algorithms for, 330–335
attributes and, 305–313, 330–335, 339–340
Boyce–Codd normal form and, 313–316, 330–333
dependency preservation and, 315–316, 329
fourth normal form and, 339–341
functional dependencies and, 308–313, 330–341
higher normal forms and, 319
keys and, 309–312
lossless, 307–308, 307n1, 312–313
lossy, 307
multivalued dependencies and, 336–341
normalization theory and, 308
notational conventions and, 309
relational database design and, 305–313, 330–341
third normal form and, 317–319, 333–335
decomposition rule, 321
decompression, 613, 615n8
decorrelation, 777–778
DEC Rdb, 26
deduplication, 523
deep learning, 546
deep neural networks, 546
de facto standards, 1237
default values
classifiers and, 545
privileges and, 167
setting reference field to, 150
user-defined types and, 159
deferred integrity constraints, 151
deferred-modification technique, 915
deferred view maintenance, 779, 1215–1216
degree of relationship sets, 249
degree-two consistency, 880–881
delete authorization, 14
deletion
B+-trees and, 641, 645–649
concurrency control and, 857–858
database modification and, 108–110
hashing and, 1190, 1194–1195, 1198
integrity constraints and, 150
LSM trees and, 1178–1179
of messages, 1110
ordered indices and, 624, 631–632
privileges and, 166–167
R-trees and, 1189
shipping SQL statements to database and, 187
SQL schema definition and, 69, 71
transactions and, 801, 826
triggers and, 208–209
tuples and, 108–110, 613
views and, 142
deletion entries, 668, 1178–1179
delta relation, 211
demand-driven pipeline, 726–728
denial-of-service attacks, 502
denormalization, 346
dense indices, 626–628, 630–631
dependency of transactions, 819
dependency preservation, 315–316, 328–330
derived attributes, 251,
desc expression, 84
descriptive attributes, 248
descriptive patterns, 541
DES (Data Encryption Standard), 448
design. ***See*** **database design**
destination-driven architecture, 522
dicing, 530
dictionary attacks, 449
differentials, 780
digital certificates, 451–453
digital ledgers, 1251, 1252
digital signatures, 451, 1257, 1261
digital video disks (DVDs), 560–561
dimension attributes, 524
dimension tables, 524
direct-access storage, 561
directed acyclic graphs (DAGs), 499, 506–507, 1071–1072
directory access protocols, 1084, 1240–1243
directory information trees (DITs), 1242, 1243
directory systems, 1020, 1084–1086, 1240–1243
dirty blocks, 928–929
dirty page table, 941–947
dirty writes, 822
disable trigger, 210
disambiguation, 549
disconnected operation, 427–428
discretized streams, 508
discriminator attributes, 259
disjoint generalization, 279, 290
disjoint specialization, 272, 275
disjoint subtrees, 847
disjunctive selection, 699, 700, 762
disk arms, 565
disk-arm–scheduling, 578–579
disk blocks, 566–567, 577–580
disk buffer, 578, 910
disk controllers, 565
disk failure, 908
distinct types, 90, 92, 98–100,

Telegram: @uni_k

www.konkur.in

distinguished name (DN), 1241–1242
distributed autonomous organizations (DAOs), 1272, 1272n7
distributed consensus problem, 1106–1107, 1151
distributed databases
architecture of, 22, 986–989, 1098–1100
autonomy and, 988
Big Data and, 473, 480–481
commit protocols and, 1100–1110
concurrency control and, 990, 1105, 1111–1120
consensus in, 1106–1107, 1150–1161, 1266, 1267
directory systems and, 1020, 1084–1086, 1240–1243
failure and, 1104
federated, 988, 1076–1077, 1132–1133
file systems in, 472–475, 489, 1003, 1019–1022
global transactions and, 988, 1098, 1132
heterogeneous, 988, 1132
homogeneous, 988
leases and, 1115–1116
local transactions and, 988, 1098, 1132
locks and, 1111–1116
nodes and, 987
partitions and, 1104–1105
persistent messaging and, 1108–1110, 1137
query optimization and, 1084
query processing and, 1076–1086
recovery and, 1105
replication and, 987, 1121–1128
sharing data and, 988
sites and, 986
snapshot isolation and, 1131–1132
timestamps and, 1116–1118
transaction processing in, 989–990, 1098–1100
validation and, 1119–1120
distributed file systems, 472–475, 489, 1003, 1019–1022
distributed hash tables, 1013
distributed-lock manager, 1112
distributed query processing, 1076–1086
data integration from multiple sources, 1076–1078
directory systems and, 1084–1086
join location and join ordering in, 1081–1082
across multiple sources, 1080–1084
optimization and, 1084
schema and data integration in, 1078–1080
semijoin strategy and, 1082–1084
DITs (directory information trees), 1242, 1243
Django framework, 382, 419–421, 433–435, 1240
DKNF (domain-key normal form), 341
DML. *See* data-manipulation language
DML compiler, 20
DN (distinguished name), 1241–1242
DNS (Domain Name Service) system, 1084, 1085
Docker, 995
document databases, 3
Document Object Model (DOM), 423
document stores, 477, 1023
domain constraints, 13–14, 146
domain-key normal form (DKNF), 341
Domain Name Service (DNS) system, 1084, 1085
domain of attributes, 39–40, 249
double buffering, 725
double-pipelined hash-join, 731
double-pipelined join technique, 730–731
double-spend transactions, 1261–1262, 1264
downgrade, 843
drill down, 531, 540
DriverManager class, 186
drop index, 165, 664
drop schema, 163
drop table, 69, 71, 190
drop trigger, 210
drop type, 159
dumping, 930–931
duplicate elimination, 719–720, 1049
durability
defined, 800
one-safe, 933
remote backup systems and, 933–934
storage structure and, 804–805
of transactions, 20–21, 800–807
two-safe, 934
two-very-safe, 933
DVDs (digital video disks), 560–561
dynamic handling of join skew, 1048
dynamic hashing, 661, 1195–1203
dynamic-programming algorithm, 767
dynamic repartitioning, 1010–1013
dynamic SQL, 66, 184, 201
Dynamo, 477, 489, 1024

Eclipse, 416
e-commerce, streaming data and, 501
edge switches, 977
efficiency of databases, 1, 2, 5, 9
e-government, 1277
elasticity, 992, 1010, 1024
election algorithms, 1147
elevator algorithm, 578
embedded databases, 198, 962
embedded multivalued dependencies, 341
embedded SQL, 66, 184, 197–198, 965, 1269

Telegram: @uni_k

www.konkur.in

encryption
Advanced Encryption Standard, 448, 449
applications of, 447–453
asymmetric-key, 448
authentication and, 450–453
blockchain databases and, 1260–1261
challenge-response system and, 451
database support and, 449–450
dictionary attacks and, 449
digital certificates and, 451–453
digital signatures and, 451
nonrepudiation and, 451
prime numbers and, 449
private-key, 1260–1261
public-key, 448–449, 1260–1261
Rijndael algorithm and, 448
symmetric-key, 448
techniques of, 447–449
end transaction operation, 799
end-user information, 443
enterprise information, database applications for, 2–4
entities, 243, 244, 247–248
entity group, 1031
entity recognition, 549
entity-relationship (E-R) diagrams
aggregation and, 279
alternative notations for modeling data, 285–291
common mistakes in, 280–281
complex attributes and, 265–267
defined, 244
entity sets and, 245–246, 265–268
generalization and, 278–279
participation illustrated by, 255
reduction to relational schema, 264–271, 277–279
relationship sets and, 247–250, 268–271
Unified Modeling Language and, 289–291
for university enterprise, 263–264
with weak entity set, 260
entity-relationship (E-R) model, 244–291
aggregation and, 276–277
alternative notations for modeling data, 285–291
atomic domains and, 342–343
attributes and, 245, 248–252, 274–275, 281–282, 342–343
constraints and, 253–256, 275–276
database design and (*see* database design)
design issues and, 279–285
development of, 244
diagrams (*see* entity-relationship diagrams)
entity sets and, 244–246, 261–264, 281–283
extended features, 271–279
generalization and, 273–274
mapping cardinalities and, 252–256
normalization and, 344–345
overview, 8
primary keys and, 256–260
redundancy and, 261–264
relationship sets and, 246–249, 282–285
schemas and, 244, 246, 269–270, 277–279
specialization and, 271–273
Unified Modeling Language and, 288–291
entity sets
alternative notations for, 285–291
attributes and, 245, 265–267, 281–282
defined, 245
design issues and, 281–283
entity-relationship diagrams and, 245–246,
entity-relationship (E-R) model and, 244–246, 261–264, 281–283
extension of, 245
hierarchies of, 273, 275
identifying, 259
primary keys and, 257
properties of, 244–246
relationship sets and, 246–249, 282–283
removing redundancy in, 261–264
representation of, 265–268
strong, 259, 265–267
subclass, 274
superclass, 274
Unified Modeling Language and, 288–291
value and, 245
weak, 259–260, 267–268
entries, 1241
equality-generating dependencies, 337
equi-depth histograms, 759
equi-joins, 704, 707–713, 718, 722, 730, 1043
equivalence
conflict, 815, 815n2
cost analysis and, 771
enumeration of expressions, 755–757
join ordering and, 754–755
relational algebra and, 58, 747–757
transformation examples for, 752–754
view, 818, 818n4
equivalence rules, 747–752, 754, 771
equivalent queries, 58
equi-width histograms, 759
erase block, 568
E-R diagrams. *See* entity-relationship diagrams
E-R model. *See* entity-relationship (E-R) model
escape, 83
Ethereum, 1258, 1262, 1265, 1267–1272, 1274

Telegram: @uni_k

www.konkur.in

ETL (extract, transform and load) tasks, 520, 524
evaluation primitive, 691
eventual consistency, 1016, 1139
every function, 96
evicted blocks, 605
exactly-once semantics, 1074
except all, 89, 97
except clause, 216
except construct, 102
exception conditions, 202
exceptions, 187
except operation, 88–89
exchange-operator model, 1055–1057
exclusive locks
biased protocol and, 1124
concurrency control and, 835–843, 888, 892, 893
degree-two consistency and, 880
graph-based protocols and, 846–847
multiple granularity and, 854–855
multiversion, 871
recovery systems and, 916
transactions and, 825, 928
exclusive-mode locks, 835, 842
EXEC SQL, 197
execute privilege, 169–170
execution skew, 1007, 1008, 1043
existence bitmaps, 1184
existence dependence, 259
exists construct, 101, 102, 108
expiration of leases, 1115
explain command, 746
explicit locks, 854
extendable hashing, 661, 1195, 1196
Extensible Markup Language. ***See*** **XML**
extension of entity sets, 245
extent (blocks), 579
external consistency, 1131
external data, 1077
external language routines, 203–206
external sorting, 701
external sort-merge algorithm, 701–704
extract, transform and load (ETL) tasks, 520, 524
extraneous attributes, 325
extraneous functional dependencies, 324

factorials, 811
fact tables, 524
failed transactions, 806, 907, 909
fail-stop assumption, 908, 1267
failure recovery, 21
false cycles, 1114–1115
false positives, 1083
false values, 96
fanout, 635
fat-tree topology, 977
fault tolerance
blockchain databases and, 1276
geographic distribution and, 1027
interconnection networks and, 978
MapReduce paradigm and, 1060–1061
in query-evaluation plans, 1059–1061
replicated state machines and, 1158–1161
shared-disk architecture and, 984
with streaming data, 1074–1076
updates and, 1138
fault-tolerant key-value store, 1160
fault-tolerant lock manager, 1160
federated distributed databases, 988, 1076–1077, 1132–1133
fetching. ***See also*** **information retrieval**
advanced SQL and, 187–188, 193, 195–197, 202, 205, 222
application design and, 421–427, 431, 437, 1218, 1229
by buffer manager, 19
data warehousing and, 526
large-object types and, 156, 158
prefetching, 969
storage and, 567, 572, 587
fiat currencies, 1252, 1273
Fiber Channel FC interface, 563
Fiber Channel Protocol, 978
fifth normal form, 341
file headers, 590–591
file manager, 19
file organization, 588–602
blobs, 156, 193, 594, 652
blocks and, 579, 588
B^+-tree, 595, 650–652, 697, 697n4
clobs, 156, 193, 594, 652
distributed, 472–475, 489, 1003, 1019–1022
fixed-length records and, 589–592
hash, 595, 659
heap file organization, 595–597
indexing and (*see* indices)
journaling systems, 610
large objects and, 594–595
multitable clustering, 595, 598–601
null values and, 593
partitioning and, 601–602
pointers and, 588, 591, 594–598, 601
reorganization, 598
sequential, 595, 597–598
variable-length records and, 592–594
file-processing systems, 5–8
file scans, 695–697, 704–707, 727
file system consistency check, 610
financial sector, database applications for, 3, 1279
fine-grained locking, 947
fine-grained parallelism, 963, 970
firm deadlines, 894
first committer wins, 874
first normal form (1NF),

Telegram: @uni_k

www.konkur.in

first updater wins, 874–875
five minute rule, 1229
fixed-length records, 589–592
fixed point of recursive view definition, 217
flash-as-buffer approach, 1229
flash memory, 567–570
cost of, 560
erase block and, 568
hybrid, 569–570
indexing on, 656–657
LSM trees and, 1182
NAND, 567–568
NOR, 567
wear leveling and, 568
flash translation layer, 568
flatMap function, 496–498
flexible schema, 366
FlinkCEP, 504
float, 67
Flutter framework, 428
followers, 1155
forced output, 607, 912
force policy, 927
for clause, 534
for each row clause, 207–210, 212
for each statement clause, 76, 209–210
foreign-currency exchange, 1277
foreign keys, 45–46, 69–70, 148–150, 267–268, 268n5
foreign tables, 1077
forking, 1257, 1258, 1263
formal standards, 1237
fourth normal form (4NF), 336, 339–341
fragment-and-replicate joins, 1046–1047, 1062
fragmentation, 579
free lists, 591
free-space maps, 596–597
from clause
aggregate functions and, 91–96
basic SQL queries and, 71–79
on multiple relations, 74–79
in multiset relational algebra, 97
null values and, 90
query optimization and, 775–777
rename operation and, 79, 81–82
set operations and, 85–89
on single relation, 71–74
string operations and, 82–83
subqueries and, 104–105
full nodes, 1256, 1268
full outer joins, 132–136, 722
functional dependencies
algorithms for decomposition using, 330–335
attribute set closure and, 322–324
augmentation rule and, 321
axioms and, 321
Boyce–Codd normal form and, 313–316, 330–333
canonical cover and, 324–328
closure of a set, 320–324
decomposition rule and, 321
dependency preservation and, 315–316, 328–330
extraneous, 324
higher normal forms and, 319
keys and, 309–312
lossless decomposition and, 312–313
multivalued, 336–341
notational conventions and, 309
pseudotransitivity rule and, 321
reflexivity rule and, 321
in schema design, 145
theory of, 320–330
third normal form and, 317–319, 333–335
transitivity rule and, 321
trivial, 311
union rule and, 321
functionally determined attributes, 322–324
functional query language, 47
functions. ***See also specific functions***
declaring, 199–201
external language routines and, 203–206
hash (*see* hash functions)
language constructs for, 201–203
syntax and, 199, 201–205
writing in SQL, 198–206
fuzzy checkpoints, 922, 930, 941
fuzzy dump, 931
fuzzy lookup, 523

gas concept for transactions, 1270–1271
GAV (global-as-view) approach, 1078–1079
generalization
attribute inheritance and, 274–275
bottom-up design and, 273
disjoint, 279, 290
entity-relationship (E-R) model and, 273–274
overlapping, 279, 290
partial, 275
representation of, 278–279
subclass set and, 274
superclass set and, 274
top-down design and, 273
total, 275
Generalized Search Tree (GiST), 670
genesis blocks, 1254–1255
geographically distributed storage, 1026–1027
geographic data
applications of, 391–392
examples of, 387
overlays and, 393
raster data, 392
representation of, 392–393
subtypes of, 390
topographical, 393
vector data, 392–393
geographic information systems, 387
geometric data, 388–390
getColumnCount method, 191–192
getConnection method, 186, 186n1
getFloat method, 188

Telegram: @uni_k

www.konkur.in

GET method, 440
getString method, 188
GFS. *See* Google File System
GiST (Generalized Search Tree), 670
Glassfish, 416
global-as-view (GAV) approach, 1078–1079
global indices, 1017–1019
Global Positioning System (GPS), 1130
global schema, 1076, 1078–1079
global transactions, 988, 1098, 1132
global wait-for graphs, 1113–1114
Google
application design and, 406–408, 410, 428–429
Bigtable, 477, 479–480, 668, 1024
PageRank from, 385–386, 493, 510
Pregel system developed by, 511
Spanner, 1160–1161
Google File System (GFS), 473, 474, 1020, 1022
Google Maps, 393
GPS (Global Positioning System), 1130
GPUs (graphics processing units), 1064
grant command, 170
granted by current role, 172–173
grant privileges, 166–167, 170–171
graph-based protocols, 846–848
graph databases, 508–511
graphics processing units (GPUs), 1064
GraphX, 511
group by clause, 92–96, 105, 142, 221, 227–230
group by construct, 534, 536–537
group by cube, 536
group by rollup, 537
group-commit technique, 925
grouping function, 536–537
grouping sets construct, 230, 538
growing phase, 841, 843
Gustafson's law, 974

hackers. *See* security
Hadoop File System (HDFS), 473–475, 489–493, 971, 1020–1022
Halloween problem, 785
handlers, 202
hard deadlines, 894
hard disk drives (HDDs). *See* magnetic disks
hard disks, 26
hard forks, 1257, 1258
hardware RAID, 574–576
hardware threads, 982
hardware tuning, 1227–1230
harmonic mean, 1231
hash file organization, 595, 659
hash functions
Bloom filters and, 1175–1176
closed, 659, 1194
collision resistant, 1259–1260
consistent, 1013
cryptographic, 1253, 1259–1263, 1265
defined, 624, 659
deletion, 1190, 1194–1195, 1198
dynamic, 661, 1195–1203
extendable, 661, 1195, 1196
insertion, 1194–1195, 1197–1202
irreversibility of, 1260
joins and, 1045
linear, 661, 1203
lookup, 1197, 1198, 1202–1203
open, 1194
partitioning and, 1045
passwords and, 1260n4
queries and, 624, 1197–1202
static, 661, 1190–1195, 1202–1203
updates and, 624, 1197–1202
hash indices
bucket overflow and, 659–660, 1194–1195
comparison with ordered indices, 1203
data structure and, 1195–1196
defined, 624
dynamic hashing and, 661, 1195–1203
extendable hashing and, 661
file organization and, 595, 659
insufficient buckets and, 1194
linear hashing and, 661, 1203
in main memory, 658–659
overflow chaining and, 659–660
skew and, 660, 1194
static hashing and, 661, 1190–1195, 1202–1203
tuning of, 1215
hash join
basics of, 712–714
build input and, 713
cost of, 715–717
hybrid, 717–718
overflows and, 715
pipelining and, 728–731
probe input and, 713
query optimization and, 769, 771
query processing and, 712–718, 786, 1063
recursive partitioning and, 714–715
skewed partitioning and, 715
hash partitioning, 476, 1005–1007
hash-table overflow, 715
hash trees. *See* Merkle trees
having clause, 95–96, 104–105, 142
HBase, 477, 480, 489, 668, 971, 1024, 1028–1031
HDDs (hard disk drives). *See* magnetic disks
HDFS. *See* Hadoop File System
head-disk assemblies, 565
health care, blockchain applications for, 1277–1278
heap files, 595–597, 1203
heart-beat messages, 1147
heterogeneous distributed

Telegram: @uni_k

www.konkur.in

heuristics, 766, 771–774, 786, 1189
Hibernate system, 382, 431–433, 1240
hierarchical architecture, 979, 980, 986
hierarchical clustering, 548
hierarchical data models, 26
hierarchies
cross-tabulation and, 532
on dimensions, 531
of entity sets, 273, 275
relational representation of, 533
transitive closures on, 214
high availability, 907, 931–933, 987, 1121
high availability proxy, 932–933
higher-level locks, 935–936
higher normal forms, 319
histograms
attributes and, 758–760
distribution approximated by, 543
equi-depth, 759
equi-width, 759
examples of, 758–759, 1009
join size estimation and, 763–764
percentile-based, 223
random samples and, 761
range-partitioning vectors and, 1009
selection size estimation and, 760
Hive, 494, 495, 500
HOLAP (hybrid OLAP), 535
Hollerith, Herman, 25
homogeneous distributed databases, 988
hopping window, 505
horizontal partitioning, 1004, 1216–1217
host language, 16, 197
hot-spare configuration, 933
hot swapping, 575
householding, 523
HTML. *See* HyperText Markup Language
HTTP. *See* HyperText Transfer Protocol
hybrid disk drives, 569–570
hybrid hash join, 717–718
hybrid merge-join algorithm, 712
hybrid OLAP (HOLAP), 535
hybrid row/column stores, 615
hypercubes, 976–977
Hyperledger Fabric, 1269, 1271
hyperlinks, 385–386
HyperText Markup Language (HTML)
application design and, 404, 406–408, 426
client-side scripting and, 421
Java Server Pages and, 417–418
security and, 439, 440
server-side scripting and, 416–418
stylesheets and, 408
web sessions and, 408–411
HyperText Transfer Protocol (HTTP)
application design and, 405–413
connectionless nature of, 409–410
man-in-the-middle attacks and, 442
Representation State Transfer and, 426
security and, 440, 452
hyper-threading, 982
hypervisor, 994

IBM DB2
advanced SQL and, 206
history of, 26
limit clause in, 222
query optimization and, 774, 783
Spatial Extender, 388
statistical analysis and, 761
trigger syntax and, 212
types and domains supported by, 160
ICOs (initial coin offerings), 1272
IDEF1X standard, 288
IDE (integrated development environment), 416
idempotent operations, 937
identifiers
block, 474–475, 1020
data-item, 913
indices and, 700
log records and, 913
query processing and, 700
selection and, 700
transaction, 913
identifying entity sets, 259
identifying relationship, 259–260
identity declaration, 1226
identity specification, 161
identity theft, 447
IDF (inverse document frequency), 384
IEEE (Institute of Electrical and Electronics Engineers), 1237
if clauses, 212
if-then-else statements, 202
immediate-modification technique, 915
immediate view maintenance, 779, 1215–1216
imperative query language, 47
implicit locks, 854–855
incompleteness in database design, 243–244
inconsistent data, 6
inconsistent state, 802, 803, 812
in construct, 99
incremental view maintenance, 779–782
increment lock, 892–893
increment operation, 892
independent parallelism, 1054–1055
indexed nested-loop join, 707–708, 728
index entries, 626
indexing strings, 653
index-locking protocol, 860
index-locking technique, 860
index records, 626
index scans, 696, 698–699, 769,

Telegram: @uni_k

www.konkur.in

index-sequential files, 625, 634–635
indices, 623–676, 1175–1203
access time and, 624, 627–628
access types and, 624
basic concepts related to, 623–624
bitmap (*see* bitmap indices)
Bloom filters and, 667, 1175–1176, 1181
B^+-tree (*see* B^+-trees)
buffer trees and, 668–670
bulk loading of, 653–655
clustering, 625, 632–633, 695, 697–698
comparisons and, 698–699
composite, 700
concurrency control and, 884–887
covering, 663
creation of, 664–665, 884–885
defined, 19
definition in SQL, 164–165, 664–665
deletion time and, 624, 631–632, 641, 645–649
dense, 626–628, 630–631
on flash storage, 656–657
Generalized Search Tree, 670
global, 1017–1019
hash (*see* hash indices)
identifiers and, 700
insertion time and, 624, 630–631, 641–645, 647, 649
inverted, 721
key-value stores and, 1028
linear search and, 695
local, 1017
LSM trees and, 666–668, 1176–1182, 1215
in main memory, 657–658
materialized views and, 783
multilevel, 628–630
multiple-key access and, 633–634, 661–663
nonclustering, 625, 695
ordered (*see* ordered indices)
parallel, 1017–1019
performance tuning and, 1215
pointers and, 700
primary, 625, 695, 1017–1018
query processing and, 695–697
record relocation and, 652–653
search keys and, 624–634
secondary, 625, 632–633, 652–653, 695–698, 1017–1019
selection operation and, 695–697, 783
sequential, 625, 634–635
sorting and, 701–704
space overhead and, 624, 627–628, 634, 1202
sparse, 626–632
spatial data and, 672–675, 1186–1190
SQL DDL and, 67
stepped-merge, 667, 1179–1181
of temporal data, 675–676
updates and, 630–632
write-optimized structures, 665–670
in-doubt transactions, 1105
infeasibility, 1259–1260
Infiniband standard, 978
information extraction, 549
information retrieval. ***See also*** **queries**
defined, 382
keywords and, 383
measuring effectiveness of, 386
PageRank and, 385–386
precision and, 386
recall and, 386
relevance ranking and, 383–386
stop words and, 385
structured data queries and, 386–387
TF-IDF approach and, 384–385
infrastructure-as-a-service model, 991
Ingres system, 26
inheritance
attribute, 274–275
multiple, 275
single, 275
tables and, 379–380
types and, 378–379
initial coin offerings (ICOs), 1272
initialization vector, 449
initially deferred integrity constraints, 151
inner joins, 132–136, 771
inner relation, 704
insert authorization, 14
insertion
algorithm for, 1180–1181
B+-trees and, 641–645, 647, 649
concurrency control and, 857, 858
database modification and, 110–111
default values and, 156
hashing and, 1194–1195, 1197–1202
LSM trees and, 1177–1178, 1180–1181
ordered indices and, 624, 630–631
prepared statements and, 188–190
privileges and, 166–167
R-trees and, 1188–1189
shipping SQL statements to database and, 187
SQL schema definition and, 69
transactions and, 801, 826
views and, 141–143
instances, 12, 309, 547. ***See also*** **training instances**
instead of feature, 143
Institute of Electrical and Electronics Engineers (IEEE), 1237
insurance claims, 1278
integrated development environment (IDE), 416

Telegram: @uni_k

www.konkur.in

add, 146
alter table and, 146
assertions and, 152–153
assigning names to, 151
authorization, 14
check clause and, 147–149, 152–153
create table and, 146–149
deferred, 151
domain, 13–14
examples of, 145
in file-processing systems, 6
foreign keys and, 148–150
functional dependencies (*see* functional dependencies)
not null, 146, 150
primary keys and, 147, 148
referential, 14, 46, 149–153, 207–208, 800
schema diagrams of, 46–47
on single relation, 146
spatial, 391
SQL and, 14–15, 66, 145–153
unique, 147
user-defined types and, 159–160
violation during transactions, 151–152
integrity manager, 19
Intel, 569, 1064
intention-exclusive (IX) mode, 855
intention lock modes, 855
intention-shard (IS) mode, 855
interconnection networks, 975–979
interesting sort order, 770
interference, 974, 1232
intermediate keys, 1050
intermediate SQL, 125–173
authorization and, 165–173
create table extensions and, 162
data/time types in, 154
default values and, 156
generating unique key values and, 160–161
index definition in, 164–165
integrity constraints and, 145–153
join expressions and, 125–136
large-object types and, 156, 158
roles and, 167–169
schemas, catalogs, and environments, 162–163
transactions and, 143–145
type conversion and formatting functions, 155–156
user-defined types and, 158–160
views and, 137–143
internal nodes, 635
International Organization for Standardization (ISO), 65, 1237, 1241
Internet. *See* World Wide Web
Internet of Things (IoT), 470, 1278
interoperation parallelism, 1040, 1052–1055
interquery parallelism, 1039
intersect all, 88, 97
intersection of bitmaps, 671, 1183
intersection operation, 750, 782
intersect operation, 54–55, 87–88
interval data type, 154
intra-node partitioning, 1004
intraoperation parallelism
aggregation and, 1049
defined, 1040
duplicate elimination and, 1049
map and reduce operations and, 1050–1052
parallel external sort-merge and, 1042–1043
parallel join and, 1043–1048
parallel sort and, 1041–1043
projection and, 1049
range-partitioning sort and, 1041–1042
selection and, 1049
intraquery parallelism, 1039
invalidation reports, 1141
invalidation timestamps,
inverse document frequency (IDF), 384
inverted indices, 721
I/O operations per second (IOPS), 567, 577, 578, 693n3
I/O parallelism
hashing and, 1005–1007
partitioning techniques and, 1004–1007
range scheme and, 1005, 1007
round-robin scheme and, 1005, 1006
IoT (Internet of Things), 470, 1278
irrefutability, 1257, 1259
irreversibility, 1260
IS (intention-shard) mode, 855
is not null, 89, 90
is not unknown, 90
is null, 89
ISO (International Organization for Standardization), 65, 1237, 1241
isolation
atomicity and, 819–821
cascadeless schedules and, 820–821
concurrency control and, 803–804, 807–812, 823
of data, 6
defined, 800
factorials and, 811
improved throughput and, 808
inconsistent state and, 803
levels of, 821–826
locking and, 823–825
multiple versions and, 825–826
read committed, 1225
recoverable schedules and, 819–820
resource allocation and, 808
serializability and, 821–826
snapshot (*see* snapshot isolation)

Telegram: @uni_k

www.konkur.in

of transactions, 800–804, 807–812, 819–826
utilization and, 808
wait time and, 808–809
is unknown, 90
iteration, 201–202, 214–216
iterators, 727
IX (intention-exclusive) mode, 855

Jakarta Project, 416
Java. *See also* JDBC (Java Database Connectivity)
advanced SQL and, 183, 197, 199, 205
application design and, 16, 417
metadata and, 191–193
object-oriented programming and, 377
object-relational mapping system for, 382
Resilient Distributed Dataset and, 496–498
Unified Modeling Language and, 289
Java Database Connectivity. *See* JDBC
Java 2 Enterprise Edition (J2EE), 416
JavaScript
application design and, 404–405, 421–426
input validation and, 422–423
interfacing with web services, 423–426
Representation State Transfer and, 426, 427
responsive user interfaces and, 423
security and, 439
JavaScript Object Notation (JSON)
applications for use, 368–369
defined, 368
emergence of, 27
encoding results with, 426
example of, 368, 369
flexibility of, 367–368
key-value stores and, 1024
mapping to data models, 480
as semi-structured data model, 8, 27
SQL in support of, 369–370
for transferring data, 423
Java Server Pages (JSP)
application design and, 405, 417–418
security and, 440
server-side scripting and, 417–418
servlets and, 417–418
Java Servlets, 411–416, 419, 424
JBoss, 416
JDBC (Java Database Connectivity), 184–193
blob column and, 193
caching and, 435–436
callable statements and, 190–191
clob column and, 193
connecting to database, 185–186
corresponding interface defined by, 17
exception and resource management, 187
metadata features and, 191–193
prepared statements and, 188–190
protocol information, 186
retrieving query results, 187–188
shipping SQL statements to, 186–187
updatable result sets and, 193
web sessions and, 409
join conditions, 130–131
join dependencies, 341
join operation, 52–53
joins
ant-ijoin operation, 108, 776
anti-semijoin operation, 776–777
broadcast, 1046
complex, 718
cost analysis and, 710–712, 767–770
equi-joins, 704, 707–713, 718, 722, 730, 1043
fragment-and-replicate, 1046–1047, 1062
full outer, 132–136, 722
hash (*see* hash join)
hybrid merge, 712
inner, 132–136, 771
inner relation of, 704
left outer, 131–136, 722
merge-join, 708–712, 1045
minimization and, 784
natural (*see* natural joins)
nested loop (*see* nested-loop join)
ordering, 754–755
outer, 57, 131–136, 722–723, 765, 782
outer relation of, 704
parallel, 1043–1048
partitioned, 714–715, 1043–1046
query processing and, 704–719, 722–723, 1081–1082
right outer, 132–136, 722
semijoin operation, 108, 776, 1082–1084
size estimation and, 762–764
skew in, 1047–1048
sort-merge-join, 708–712
spatial, 394
spatial data and, 719
streaming data and, 506
theta, 748–749
types and conditions, 130–131, 136
view maintenance and, 780
join skew avoidance, 1048
join using operation, 129–130
journaling file systems, 610
JSON. *See* JavaScript Object Notation
JSP. *See* Java Server Pages
J2EE (Java 2 Enterprise Edition), 416
jukebox systems, 561

Kafka system, 506, 507,

Telegram: @uni_k

www.konkur.in

k-d B trees, 674
KDD (knowledge discovery in databases), 540
k-d trees, 673–674
kernel functions, 545
keys
candidate, 44
cluster key, 600–601
constraints and, 258
encryption and, 448–449, 451–453
equality on, 697
foreign, 45–46, 69–70, 148–150, 267–268, 268n5
functional dependencies and, 309–312
intermediate, 1050
multiple access, 633–634, 661–663
partitioning, 475–476
primary (*see* primary keys)
reduce, 485
in relational model, 43–46
search (*see* search keys)
smart cards and, 451
superkeys, 43–44, 257–258, 309–310, 312
unique values, 160–161
key-value locking, 887
key-value maps, 366–367
key-value storage systems
Big Data and, 471, 473, 476–480
clusters and, 477
document, 477, 1023
fault tolerant, 1160
parallel, 1003, 1023–1031
replication and, 476
social-networking sites and, 471
wide-column, 1023
keyword queries, 383, 385–387, 721
killing the mutant, 1234
knowledge discovery in databases (KDD), 540
knowledge graphs, 374–375, 386–387, 549
knowledge representation, 368
Kubernetes, 995
lambda architecture, 504, 1071
language constructs, 201–203
Language Integrated Query (LINQ), 198
LANs. *See* local-area networks
large object storage, 594–595
large-object types, 156, 158
LastLSN, 943, 946
latches, 886, 928
latch-free data structure, 888–890
latency, 989
latent failure, 575
lateral clause, 105
LAV (local-as-view) approach, 1079
lazy generation of tuples, 727
lazy propagation of updates, 1122, 1136
LDAP Data Interchange Format (LDIF), 1242
LDAP (Lightweight Directory Access Protocol), 442, 1085, 1240–1243
leaders, 1155
leaf nodes, 635–656, 665–669, 673, 674
learners, 1148, 1153
leases, 1115–1116, 1147
least recently used (LRU) strategy, 605, 607
ledgers, digital, 1251, 1252
left-deep join orders, 773
left outer join, 131–136, 722
legacy systems, 1035–1036
legal instance, 309
LevelDB, 668
Lightning network, 1275
light nodes, 1256, 1268
Lightweight Directory Access Protocol (LDAP), 442, 1085, 1240–1243
like operator, 82–83
limit clause, 222
linear hashing, 661, 1203
linearizability, 1121
linear probing, 1194
linear regression, 546
linear scaleup, 972–973
linear search, 695
linear speedup, 972
line segments, 388–390
linestrings, 388, 390
Linked Data project, 376, 1080
LINQ (Language Integrated Query), 198
load balancers, 934–935, 1214
load barrier, 983
local-area networks (LANs), 977, 978, 985, 989
local-as-view (LAV) approach, 1079
local autonomy, 988
local indices, 1017
local schema, 1076
localtimestamp, 154
local transactions, 988, 1098, 1132
local wait-for graphs, 1113
lock conversions, 843
lock-free data structure, 890
locking protocols
B-link tree, 886
concurrency control and, 835–848
defined, 839
distributed lock-manager, 1112
graph-based, 846–848
implementation of, 844–846
index, 860
key-value, 887
multiple granularity, 856–857
next-key, 887
single lock-manager, 1111
transactions and, 823–825
two-phase, 841–844, 871–872, 1129–1131
lock managers, 844–845, 965, 1160
lock point, 841
locks
adaptive granularity and, 969–970
caching and, 969
call back and, 969
compatibility function and,

Telegram: @uni_k

www.konkur.in

deadlocks (*see* deadlocks)
de-escalation and, 970
distributed databases and, 1111–1116
escalation and, 857, 1227
exclusive (*see* exclusive locks)
explicit, 854
false cycles and, 1114–1115
fine-grained, 947
granting of, 836, 840–841
growing phase and, 841, 843
higher-level, 935–936
implicit, 854–855
increment, 892–893
intention modes and, 855
leases, 1115–1116, 1147
logical undo operations and, 935–941
lower-level, 935–936
multiple granularity and, 853–857
multiversion schemes and, 871–872, 1129–1131
predicate, 828, 861, 861n1
recovery systems and, 935–941
request operation and, 835–841, 844–846, 849–853, 886
shared, 825, 835, 854, 880, 1124
shrinking phase and, 841, 843
starvation and, 853
timeouts and, 850–851
timestamps and, 861–866
transaction servers and, 965–968
true matrix value and, 836
wait-for graph and, 851–852, 1113–1114

lock table, 844, 845, 967
log disks, 610
log force, 927
logical clock, 1118
logical counter, 862
logical-design phase, 18, 242
logical error, 907
logical level of abstraction, 9–12
logical logging, 936
logically implied schema, 320
logical operations
concurrency control and, 940–941
consistency and, 936–937
defined, 935
early lock release and, 935–941
idempotent, 937
log records and, 936–940
rollback and, 937–939
logical routing of tuples, 506–507, 1071–1073
logical schema, 12–13, 1223–1224
logical undo operations, 935–941
log of transactions, 805
log processing, 486–488
log records
ARIES and, 941–946
buffering and, 926–927
checkpoint, 943
compensation, 922, 942, 945
database modification and, 915–916
force/no-force policy and, 927
identifiers and, 913
logical undo operations and, 936–940
old/new values and, 913
physical, 936
recovery systems and, 913–919, 926–927
redo operation and, 915–919
steal/no-steal policy and, 927
undo operation and, 915–919
write-ahead logging rule and, 926–929
log sequence number (LSN), 941–946
log-structured merge (LSM) trees, 1176–1182
basic, 1179
Bloom filters and, 1181
deletion and, 1178–1179
for flash storage, 1182
insertion into, 1177–1178, 1180–1181
levels of, 1176
lookup and, 1181
parallel key-value stores and, 1028
performance tuning and, 1215
rolling merges and, 1178
stepped-merge indices and, 1179–1181
updates and, 1178–1179
write-optimized structure of, 666–668
log writer process, 965
long-duration transactions, 890–891
lookup
in blockchain databases, 1268
Bloom filters and, 667, 1181
concurrency control and, 884–887
data-storage systems and, 480
fuzzy, 523
hashing and, 1197, 1198, 1202–1203
indices and, 630, 637, 640–641, 645–651, 656–661, 666–669, 676
LSM trees and, 1181
query optimization and, 769
query processing and, 698, 707
lossless decomposition, 307–308, 307n1, 312–313
lossy decompositions, 307
lost update problem, 1016
lost updates, 874
lower-level locks, 935–936
loyalty programs, 1278
LRU (least recently used) strategy, 605, 607
LSM trees. *See* log-structured merge trees
LSN (log sequence number), 941–946

machine-learning algorithms, 495
magnetic disks, 563–567
access time and, 561, 566, 567
blocks and, 566–567
capacity of, 560
checksums and, 565
crashes and, 565

Telegram: @uni_k

www.konkur.in

disk controller and, 565
failure classification and, 908
hybrid, 569–570
mean time to failure and, 567
performance measures of, 565–567
physical characteristics of, 563–565
read-write heads and, 564–565
recording density and, 565
sectors and, 564–566
seek time and, 566, 566n2, 567
sizes of, 563
main memory, 559–560, 657–658
main-memory databases
accessing data in, 910
concurrency control in, 887–890
recovery in, 947–948
storage in, 588, 615–617
majority protocol, 1123–1126
management of data. ***See*** **database-management systems** (DBMSs)
man-in-the-middle attacks, 442
manufacturing, database applications for, 3, 5
many-to-many mapping, 253–255
many-to-one mapping, 252–255
map function, 483–494, 510. ***See also*** **MapReduce paradigm**
mapping cardinalities, 252–256
MapReduce paradigm, 483–494
development of, 27, 481
fault tolerance and, 1060–1061
in Hadoop, 489–493
intraoperation parallelism and, 1050–1052
log processing and, 486–488
parallel processing of tasks, 488–489
SQL on, 493–494
word count program and, 483–486, 484n2, 490–492
markup languages. ***See specific languages***
massively parallel systems, 970
master nodes, 1012, 1136
master replica, 1016
master sites, 1026
master-slave replication, 1137
master table, 1222
match clause, 509
materialization, 724–725
materialized edges, 728
materialized views
aggregation and, 781–782
defined, 140, 778
index selection and, 783
join operation and, 780
parallel maintenance of, 1069–1070
performance tuning and, 1215–1216
projection and, 780–781
query optimization and, 778–783
selection and, 780–781
view maintenance and, 140, 779–782
max function, 91–92, 105, 723, 766, 782
maximum margin line, 544
mean time between failures (MTBF), 567n3
mean time to data loss, 571
mean time to failure (MTTF), 567, 567n3
mean time to repair, 571
measure attributes, 524
mediators, 1077
memcached system, 436–437, 482
memoization, 771
memory. ***See also*** **storage**
bulk loading of indices and, 653–655
cache, 559 (*see also* caching)
data access and, 910–912
flash, 560, 567–570, 656–657
force output and, 912
magnetic-disk, 560, 563–567
main, 559–560,
non-volatile random-access, 579–580
optical, 560–561
overflows and, 715
query costs and, 697
query processing in, 731–734
recovery systems and, 910–912
storage class, 569, 588, 948
memory barrier, 983–984
merge-join, 708–712, 1045
merge-purge operation, 523
merging
duplicate elimination and, 719–720
exchange-operator model and, 1055–1057
ordered, 1056
parallel external sort-merge, 1042–1043
performance tuning and, 1222–1223
query processing and, 701–704, 708–712
random, 1056
rolling merge, 1178
Merkle-Patricia trees, 1269, 1275
Merkle trees, 1143–1146, 1268, 1269
mesh system, 976
MESI protocol, 984
message-based consensus, 1266
message delivery process, 1109–1110
metadata, 14, 191–193, 470, 602–604, 1020–1022
microservices architecture, 994
Microsoft
application design and, 417, 442
Database Tuning Assistant, 1217
query languages developed by, 538
query optimization and, 783
StreamInsight, 504
Microsoft SQL Server
advanced SQL and, 206

Telegram: @uni_k

www.konkur.in

limit clause in, 222
performance monitoring tools, 1212
performance tuning tools, 1218
procedural languages supported by, 199
snapshot isolation and, 1225
spatial data and, 388, 390
string operations and, 82
min function, 91–92, 723, 766, 782
minibatch transactions, 1227
minimal equivalence rules, 754
mining pools, 1266. ***See also*** **data mining**
minus, 88n7
mirroring, 571–573, 576, 577
mobile application platforms, 428–429
mobile phone applications, 469
models for data mining, 540
model-view-controller (MVC) architecture, 429–430
MOLAP (multidimensional OLAP), 535
MonetDB, 367, 615
MongoDB, 477–479, 482, 489, 668, 1024, 1028
monotonic queries, 218
Moore's law, 980n4
most recently used (MRU) strategy, 608–609
MRU (most recently used) strategy, 608–609
MTBF (mean time between failures), 567n3
MTTF (mean time to failure), 567, 567n3
multidimensional data, 524, 527–532
multidimensional OLAP (MOLAP), 535
multilevel indices, 628–630
multimaster replication, 1137
multiple consensus protocol, 1151
multiple granularity
concurrency control and, 853–857
hierarchy of, 854
intention-exclusive mode and, 855
intention-shared mode and, 855
locking protocol and, 856–857
request operation and, 854, 855
shared and intention-exclusive mode and, 855
tree architecture and, 853–857
multiple inheritance, 275
multiple-key access, 633–634, 661–663
multiprogramming, 809
multiquery optimization, 785–786
multiset except, 88n7
multiset relational algebra, 80, 97, 108, 136, 747
multiset types, 366
multisig instruction, 1269–1270
multitable clustering file organization, 595, 598–601
multitasking, 961, 963
multiuser systems, 962
multivalued attributes, 251, 252, 342
multivalued data types, 366–367
multivalued dependencies, 336–341
multiversion concurrency control (MVCC), 869–872
multiversion timestamp-ordering scheme, 870–871, 1118
multiversion two-phase locking (MV2PL) protocol, 871–872, 1129–1131
mutations, 1234
mutual exclusion, 965, 967
MVCC (multiversion concurrency control), 869–872
MVC (model-view-controller) architecture, 429–430
MV2PL (multiversion two-phase locking) protocol, 871–872, 1129–1131
MySQL
attributes and, 149n8
growth of, 27
joins and, 133n4
LSM trees and, 668
performance monitoring tools, 1212
string operations and, 82
unique key values in, 161

naÃ¨ve Bayesian classifiers, 543
naÃ¨ve users, 24
namenode, 475, 1020
NAND flash memory, 567–568
NAS (network attached storage), 563, 934
natural join operation, 57
natural joins, 126–136
on condition and, 130–131
conditions and, 130–131
full outer, 132–136, 722
inner, 132–136, 771
left outer, 131–136, 722
outer, 57, 131–136
right outer, 132–136, 722
navigation systems, database applications for, 3
nearest-neighbor queries, 394, 672, 674
nearness queries, 394
negation, 762
Neo4j graph database, 509, 510
nested data types, 367–368
nested-loop join
block, 705–707
indexed, 707–708, 728
parallel, 1045–1046
query optimization and, 768, 769, 771, 773
query processing and, 704–708, 713–719, 722, 786
nested subqueries, 98–107
from clause and, 104–105
with clause and, 105–106
duplicate tuples and, 103
empty relations test and, 101–102
optimization of, 774–778,

Telegram: @uni_k

www.konkur.in

scalar, 106–107
set operations and, 98–101
nesting, 854, 946
NetBeans, 416
network attached storage (NAS), 563, 934
network latency, 969
network partition, 481, 989, 989n5, 1104–1105
network round-trip time, 969
networks
deep neural, 546
interconnection, 975–979
local area, 977, 978, 985, 989
spatial, 390
wide-area, 989
neural-net classifiers, 545–546
new value, 913
next-key locking protocols, 887
next method, 188
nextval for, 1226
NFNF (non first-normal-form), 367
nodes
in blockchain databases, 1255–1257, 1263–1268
coalescing, 641, 886
datanodes, 475, 1020
defined, 468
distributed databases and, 987
failure of, 1103–1104
full, 1256, 1268
in Hadoop File System, 474, 475
internal, 635
leaf, 635–656, 665–669, 673, 674
leases and, 1115–1116
light, 1256, 1268
master, 1012, 1136
mesh architecture and, 976
multiple granularity and, 853–857
namenode, 475, 1020
nonleaf, 635–636, 642, 645–656, 663
operation, 506–507
in parallel databases, 970
primary, 1123
ring architecture and, 976
splitting of, 641–645, 886
straggler, 1060–1061
updates and, 641–647
virtual, 1009–1010
no-force policy, 927
nonbinary relationship sets, 283–285
non-blocking two-phase commit, 1161
nonces, 1265, 1271
nonclustering indices, 625, 695
nondeclarative actions, 183
non first-normal-form (NFNF), 367
nonleaf nodes, 635–636, 642, 645–656, 663
nonprocedural DMLs, 15
nonprocedural languages, 15, 16, 18, 26
nonrepudiation, 451
Non-Uniform Memory Access (NUMA), 981, 1063
nonunique search keys, 632, 637, 640, 649–650
Non-Volatile Memory Express (NVMe) interface, 562
non-volatile random-access memory (NVRAM), 579–580, 948
non-volatile storage, 560, 562, 587–588, 804, 908–910, 930–931
non-volatile write buffers, 579–580
NOR flash memory, 567
normal forms
atomic domains and, 342–343
Boyce–Codd, 313–316, 330–333, 336
domain-key, 341
fifth, 341
first, 342–343
fourth, 336, 339–341
higher, 319
join dependencies and, 341
project-join, 341
second, 316n8, 341–342, 356
third, 317–319, 333–335
normalization
in conceptual-design process, 17
denormalization, 346
entity-relationship (E-R) model and, 344–345
performance and, 346
relational database design and, 308
NoSQL systems, 28, 473, 477, 1269, 1276
no-steal policy, 927
not connective, 74
not exists construct, 101–102, 108, 218
notifications, 436
not in construct, 99, 100
not null, 69, 89–90, 142, 146, 150, 159
not operation, 89–90
not unique construct, 103
null bitmap, 593
null rejecting property, 751
null values
aggregation with, 96
attributes and, 251–252
defined, 40, 67
file organization and, 593
integrity constraints and, 145–147, 150
SQL and, 89–90
temporal data and, 347
triggers and, 209
user-defined types and, 159
NUMA (Non-Uniform Memory Access), 981, 1063
numeric, 67, 70
nvarchar, 68
NVMe (Non-Volatile Memory Express) interface, 562
NVRAM (non-volatile random-access memory), 579–580, 948
N-way merge, 702

OAuth protocol, 443
obfuscation, 1235
object-based databases
array types and, 378
complex data types and,

Telegram: @uni_k

www.konkur.in

inheritance and, 378–380
mapping and, 377, 381–382
overview, 9
reference types and, 380–381
object classes, 1242
Object Database Management Group (ODMG), 1239–1240
Object Management Group (OMG), 288
object of triples, 372
object-oriented databases (OODB), 9, 26, 377, 431, 1239–1240
object-relational databases, 377–381
defined, 377
reference types and, 380–381
table inheritance and, 379–380
type inheritance and, 378–379
user-defined types, 378
object-relational data models, 27, 376–382
object-relational mapping (ORM), 377, 381–382, 431–434, 1239–1240
observable external writes, 807
ODBC (Open Database Connectivity)
advanced SQL and, 194–197
API defined by, 194–195
application interfaces defined by, 17
caching and, 436
conformance levels and, 196–197
standards for, 1238–1239
type definition and, 196
web sessions and, 409
ODMG (Object Database Management Group), 1239–1240
off-chain transactions, 1275
offline storage, 561
OGC (Open Geospatial Consortium), 388
OLAP. *See* online analytical processing
old value, 913
OLE-DB, 1239
OLTP (online transaction processing), 4, 521, 1231–1232
OMG (Object Management Group), 288
on condition, 130–131
on delete cascade, 150, 210, 268n5
1NF (first normal form), 342–343
one-to-many mapping, 252–255
one-to-one mapping, 252–254
online analytical processing (OLAP), 527–540
aggregation on multidimensional data, 527–532
cross-tabulation and, 528–533
data cubes and, 529–530
defined, 520, 530
dicing and, 530
drill down and, 531, 540
hybrid, 535
implementation of, 535
multidimensional, 535
performance benchmarks and, 1231–1232
relational, 535
reporting and visualization tools, 538–540
rollup and, 530–531, 536–538
slicing and, 530
in SQL, 533–534, 536–538
online index creation, 884–885
online storage, 561
online transaction processing (OLTP), 4, 521, 1231–1232
on update cascade, 150
OODB. *See* object-oriented databases
open addressing, 1194
Open Database Connectivity. *See* ODBC
Open Geospatial Consortium (OGC), 388
open hashing, 1194
OpenID protocol, 443
open polygons, 388n3
open time intervals, 675
operation consistent state, 936–937
operation nodes, 506–507
operation serializability, 885
operator trees, 724, 1040
optical storage, 560–561
optimistic concurrency control, 869
optimistic concurrency control without read validation, 883, 891
optimization cost budget, 774
Oracle
advanced SQL and, 206
application server, 416
database design and, 443n6, 444–445
decode function in, 155
Event Processing, 504
GeoRaster extension, 367
history of, 26
JDBC interface and, 185, 186
keywords in, 81n3, 88n7
limit clause in, 222
nested subqueries and, 104
performance monitoring tools, 1212
performance tuning tools, 1218
procedural languages supported by, 199
query-evaluation plans and, 746
query optimization and, 773, 774, 783
reference types and, 380
set and array types supported by, 367
snapshot isolation and, 1225
Spatial and Graph, 388
statistical analysis and, 761
syntax supported by, 204, 212, 218, 218n9
transactions and, 822, 826,

Telegram: @uni_k

www.konkur.in

types and domains supported by, 160
Virtual Private Database, 173, 444–445
oracles, 1271–1272
ORC, 490, 499, 613–614
or connective, 74
order by clause, 83–84, 219–222, 534
ordered indices, 624–634
comparison with hash indices, 1203
defined, 624
dense, 626–628, 630–631
multilevel, 628–630
secondary, 625, 632–633
sequential, 625, 634–635
sparse, 626–632
techniques for, 624
updates and, 630–632
ordered merge, 1056
ORM. *See* object-relational mapping
or operation, 89–90
orphaned blocks, 1257, 1263
outer-join operation, 57, 131–136, 722–723, 765, 782
outer relation, 704
outer union operations, 229n16
outsourcing, 28
overflow avoidance, 715
overflow blocks, 598
overflow buckets, 659–660, 1194–1195
overflow chaining, 659–660
overflow resolution, 715
overlapping generalization, 279, 290
overlapping specialization, 272, 275
overlays, 393

page (blocks), 567
PageLSN, 942–945
PageRank, 385–386
page shipping, 968
parallel databases
architecture of, 22, 970–986
Big Data and, 473, 480–481
coarse-grain, 963, 970
concurrency control and, 990
defined, 480
exchange-operator model and, 1055–1057
fine-grain, 963, 970
hierarchical, 979, 980, 986
indices in, 1017–1019
interconnection networks and, 975–979
interference and, 974
interoperation parallelism and, 1040, 1052–1055
interquery parallelism and, 1039
intraoperation parallelism and, 1040–1052
intraquery parallelism and, 1039
I/O parallelism and, 1004–1007
key-value stores and, 1023–1031
massively parallel, 970
motivation for, 970–971
operator trees and, 1040
partitioning techniques and, 1004–1007
performance measures for, 971–974
pipelines and, 1053–1054
query optimization and, 1064–1070
replication and, 1013–1016
response time and, 971–972
scaleup and, 972–974
shared disk, 979, 980, 984–985
shared memory, 979–984, 1061–1064
shared nothing, 979, 980, 985–986, 1040–1041, 1061–1063
skew and, 974, 1007–1013, 1043, 1062
speedup and, 972–974
start-up costs and, 974, 1066
throughput and, 971
transaction processing in, 989–990
parallel external sort-merge, 1042–1043
parallel indices, 1017–1019
parallelism
coarse-grained, 963
data, 1042, 1057
fine-grained, 963
improvement of performance via, 571–572
independent, 1054–1055
interoperation, 1040, 1052–1055
interquery, 1039
intraoperation, 1040–1052
intraquery, 1039
I/O (*see* I/O parallelism)
Single Instruction Multiple Data, 1064
parallel joins, 1043–1048
fragment-and-replicate, 1046–1047, 1062
hash, 1045
nested-loop, 1045–1046
partitioned, 1043–1046
skew in, 1047–1048
parallel key-value stores, 1023–1031
atomic commit and, 1029
concurrency control and, 1028–1029
data representation and, 1024–1025
defined, 1003
elasticity and, 1024
failures and, 1029–1030
geographically distributed, 1026–1027
index structure and, 1028
managing without declarative queries, 1030–1031
overview, 1023–1024
performance optimizations and, 1031
storing and retrieving data, 1025–1028
support for transactions, 1028–1030
parallel processing, 437, 488–489

Telegram: @uni_k

www.konkur.in

choosing, 1066–1068
colocation of data and, 1068–1069
cost of, 1065–1066
evaluation of, 1052–1061
materialized views and, 1069–1070
space for, 1064–1065
parallel sort, 1041–1043
parameterized views, 200
parameter style general, 205
parametric query optimization, 786
parity bits, 572, 574, 577
Parquet, 490, 499
parsing
application design and, 418
bulk loads and, 1221–1223
query processing and, 689–690
partial aggregation, 1049
partial dependency, 356
partial failure, 909
partial generalization, 275
partially committed transactions, 806
partial participation, 255
partial rollback, 853
partial schedules, 819
partial specialization, 275
participation in relationship sets, 247
partitioning attribute, 479
partitioning keys, 475–476
partitioning vector, 1005
partitions
balanced range, 1008–1009
data, 989n5
defined, 1100
distributed databases and, 1104–1105
distributed file systems and, 473
dynamic repartitioning, 1010–1013
exchange-operator model and, 1055–1057
file organization and, 601–602
hash, 476, 1005–1007, 1045
horizontal, 1004, 1216–1217
intra-node, 1004
joins and, 714–715, 1043–1046
network, 481, 989, 989n5, 1104–1105
parallel databases and, 1004–1007
point queries and, 1006
query optimization and, 1065
range, 476, 1005, 1007, 1178
recursive, 714–715
of relation schema, 1216–1217
round-robin, 1005, 1006
scanning a relation and, 1006
sharding and, 473, 475–476, 1275
skew and, 715, 1007–1013
topic-partition, 1073
vertical, 1004
virtual node, 1009–1010
partition tables, 1011–1014
passwords
application design and, 403, 411, 414, 432, 450–451
dictionary attacks and, 449
distributed databases and, 1240
hash functions and, 1260n4
leakage of, 440–441
man-in-the-middle attacks and, 442
one-time, 441
single sign-on system and, 442–443
SQL and, 163, 186, 196
storage and, 602, 603
unencrypted, 414n4
path expressions, 372, 381
pattern matching, 504
Paxos protocol, 1152–1155, 1160–1161, 1267
PCIe interface, 562
pen drives, 560
performance
access time and, 561, 566–567, 578, 624, 627–628, 692
application design and, 434–437
benchmarks (*see* performance benchmarks)
of blockchain databases, 1274–1276
B+-trees and, 634, 665–666
caching and, 435–437
data-transfer rate and, 566
denormalization for, 346
improvement via parallelism, 571–572
magnetic disk storage and, 565–567
mean time to failure and, 567, 567n3
monitoring tools, 1212
parallel databases and, 971–974
parallel key-value stores and, 1031
parallel processing and, 437
response time (*see* response time)
seek time and, 566, 566n2, 567, 692, 710
sequential indices and, 634–635
testing, 1235
throughput (*see* throughput)
tuning (*see* performance tuning)
web applications and, 405–411
performance benchmarks, 1230–1234
database-application classes and, 1231–1232
defined, 1230
suites of tasks and, 1231
of Transaction Processing Performance Council, 1232–1234
performance tuning, 1210–1230
automated, 1217–1218
bottleneck locations and, 1211–1213, 1215, 1227
bulk loads and, 1221–1223
of concurrent transactions, 1224–1227

Telegram: @uni_k

www.konkur.in

horizontal partitioning of relation schema, 1216–1217
indices and, 1215
levels of, 1213–1214
materialized views and, 1215–1216
motivation for, 1210–1211
parameter adjustment and, 1210, 1213–1215, 1220, 1228, 1230
physical design and, 1217–1218
of queries, 1219–1223
RAID and, 1214, 1229–1230
of schema, 1214–1218, 1223–1224
set orientation and, 1220–1221
simulation and, 1230
tools for, 1218
updates and, 1221–1223, 1225–1227
period declaration, 157
Perl, 206
permissioned blockchains, 1253–1254, 1256–1257, 1263, 1266, 1274
persistent messaging, 990, 1016, 1108–1110, 1137
Persistent Storage Module (PSM), 201
phantom phenomenon, 827, 858–861, 877–879, 877n5, 885, 887
PHP, 405, 417, 418
physical blocks, 910
physical data independence, 9–10, 13
physical-design phase, 18, 242–243
physical equivalence rules, 771
physical level of abstraction, 9, 11, 12, 15
physical logging, 936
physical schema, 12, 13
physical storage systems, 559–580
cache memory, 559
cost per byte, 560, 561, 566n2, 569, 576
disk-block access and, 577–580
flash memory, 560, 567–570, 656–657
hierarchy of, 561, 562
indices and, 630n2
interfaces for, 562–563
magnetic disks, 560, 563–567
main memory, 559–560
optical storage, 560–561
RAID, 562, 570–577
solid-state drives, 18, 560
tape storage, 561
volatility of, 560, 562
physiological redo operations, 941
Pig Latin, 494
pin count, 605
pinned blocks, 605
pin operations, 605
pipelined edges, 728
pipeline stage, 728
pipelining, 724–731
benefits of, 726
for continuous-stream data, 731
demand-driven, 726–728
evaluation algorithms for, 728–731
implementation of, 726–728
parallelism and, 1053–1054
producer-driven, 726–728
uses for, 691–692, 724, 725
pivot attribute, 227
pivot clause, 227, 534
pivoting, 226–227, 530
pivot-table, 226–227, 528–529
PJNF (project-join normal form), 341
plan caching, 774
platform-as-a-service model, 992–993
platters, 563, 565
PL/SQL, 199, 204
pointers. ***See also*** **indices**
blockchain databases and, 1254–1255, 1261, 1269
B^+-tree (*see* B^+-trees)
concurrency control and, 886, 888, 889
query processing and, 697, 698, 700, 708
recovery systems and, 914, 945
redistribution of, 646
SQL basics and, 193, 205
storage and, 588, 591, 594–598, 601
point queries, 1006, 1190
polygons, 388–390, 388n3, 393
polylines, 388–389, 388n3
population, 547
PostgreSQL
advanced SQL and, 206
array types on, 378
concurrency control and, 873, 879
Generalized Search Tree and, 670
growth of, 27
heap file organization and, 596
JDBC interface and, 185
JSON and, 370
performance monitoring tools, 1212
PostGIS extension, 367, 388, 390
procedural languages supported by, 199
query-evaluation plans and, 746
query processing and, 692, 694, 698–699
set and array types supported by, 367
snapshot isolation and, 1225
statistical analysis and, 761
transaction management in, 822, 826
types and domains supported by, 160
unique key values in, 161
P + Q redundancy schema, 573, 574
Practical Byzantine Fault Tolerance, 1267

Telegram: @uni_k

www.konkur.in

precision, 386
precision locking, 861n1
predicate locking, 828, 861, 861n1
predicate of triples, 372
predicate reads, 858–861
prediction, 541–543, 545–546
predictive models, 4–5
preemption, 850
prefetching, 969
prefix compression, 653
Pregel system, 511
prepared statements, 188–190
presentation layer, 429
price per TPS, 1232
primary copy, 1123
primary indices, 625, 695, 1017–1018
primary keys
attributes and, 310n4
defined, 44
entity-relationship (E-R) model and, 256–260
functional dependencies and, 313
integrity constraints and, 147, 148
in relational model, 44–46
SQL schema definition and, 68–70
primary nodes, 1123
primary site, 931
primary storage, 561
prime attributes, 356
privacy, 438, 446, 1252
private-key encryption, 1260–1261
privileges
all, 166
defined, 165
execute, 169–170
granting, 166–167, 170–171
public, 167
references, 170
revoking, 166–167, 171–173
select, 171, 172
transfer of, 170–171
update, 170
probe input, 713
procedural DMLs, 15
procedural languages, 47n3, 184, 199, 204
procedures
declaring, 199–201
external language routines and, 203–206
language constructs for, 201–203
syntax and, 199, 201–205
writing in SQL, 198–206
process monitor process, 965
producer-driven pipeline, 726–728
programming languages. ***See also specific*** languages
accessing SQL from, 183–198
mismatch and, 184
object-oriented, 377
variable operation of, 184
Progressive Web Apps (PWA), 429
projection
intraoperation parallelism and, 1049
query optimization and, 764
query processing and, 720
view maintenance and, 780–781
project-join normal form (PJNF), 341
project operation, 49–50
proof-of-stake consensus, 1256, 1266
proof-of-work consensus, 1256, 1264–1266
proposers, 1148, 1152
proximity of terms, 385
PR quadtrees, 1187
pseudotransitivity rule, 321
PSM (Persistent Storage Module), 201
public blockchains, 1253, 1255, 1257–1259, 1263, 1264
public-key encryption, 448–449, 1260–1261
publish-subscribe (pub-sub) systems, 507, 1072, 1137–1139
pulling data, 727
punctuations, 503–504
pushing data, 727
put function, 477, 478
puzzle friendliness, 1265
PWA (Progressive Web Apps), 429
Python
advanced SQL and, 183, 193–194, 206
application design and, 16, 405, 416, 419
object-oriented programming and, 377
object-relational mapping system for, 382
web services and, 424

quadratic split heuristic, 1189
quads, 376
quadtrees, 392, 674, 1186–1187
queries. ***See also*** **information retrieval**
ADO.NET and, 184
basic structure of SQL queries, 71–79
on B^+-trees, 637–641, 690
caching and, 435–437
Cartesian product and, 76–79, 81, 230
compilation of, 733
continuous, 503, 731
correlated subqueries and, 101
cost of (*see* query cost)
decision-support, 521, 971
declarative, 1030–1031
defined, 15
deletion and, 108–110
equivalent, 58
evaluation of (*see* query-evaluation plans)
hash functions and, 624, 1197–1202
indices and, 623, 695–697, 707–708
insertion and, 110–111
intermediate SQL and (*see* intermediate SQL)
JDBC and, 184–193

Telegram: @uni_k

www.konkur.in

keyword, 383, 385–387, 721
languages (*see* query languages)
metadata and, 191–193
monotonic, 218
multiple-key access and, 661–663
on multiple relations, 74–79
nearest-neighbor, 394, 672, 674
nested subqueries, 98–107
null values and, 89–90
ODBC and, 194–197
optimization of (*see* query optimization)
PageRank and, 385–386
performance tuning of, 1219–1223
point, 1006, 1190
processing (*see* query processing)
programming language access and, 183–198
Python and, 193–194
range, 638, 672, 674, 1006, 1190
read only, 1039
recursive, 213–218
region, 393–394
rename operation and, 79, 81–82
ResultSet object and, 185, 187–188, 191–193, 638–639
retrieving results, 187–188
scalar subqueries, 106–107
security and, 437–446
servlets and, 411–421
set operations and, 85–89, 98–101
on single relation, 71–74
spatial, 393–394
spatial graph, 394
streaming data and, 502–506, 1070–1071
string operations and, 82–83
transaction servers and, 965
universal Turing machines and, 16
views and, 137–143

query cost
optimization and, 745–746, 757–766
processing and, 692–695, 697, 702–704, 710–712, 715–717
query-evaluation engine, 20
query-evaluation plans
choice of, 766–778
cost of, 1065–1066
defined, 691
expressions and, 724–731
fault tolerance in, 1059–1061
materialization and, 724–725
optimization and (*see* query optimization)
parallel (*see* parallel query plans)
performance tuning of, 1219–1220
pipelining and, 691–692, 724–731
relational algebra and, 690–691
resource consumption and, 694–695, 1065–1066
response time and, 694–695
role in query processing, 689, 690
viewing, 746
query-execution engine, 691
query-execution plans, 691
query languages. ***See also specific languages***
accessing from programming languages, 183–198
categorization of, 47
Cypher, 509
defined, 15, 47
in relational model, 47–48
SPARQL, 375–376
stream, 503–506
query optimization, 743–787
adaptive, 786–787
aggregation and, 764
Cartesian product and, 748, 749, 755, 763–764, 775
cost analysis and, 745–746, 757–766
defined, 20, 691, 743
distributed, 1084
equivalence and, 747–757
estimating statistics of expression results, 757–766
heuristics in, 766, 771–774, 786
hybrid hash join and, 717–718
indexed nested-loop join and, 707–708
join minimization and, 784
materialized views and, 778–783
multiquery, 785–786
nested subqueries and, 774–778, 1220
parallel databases and, 1064–1070
parametric, 786
plan choice for, 766–778
projection and, 764
relational algebra and, 743–749, 752, 755
role in query processing, 689, 690
set operations and, 764–765
shared scans and, 785–786
top-*K*, 784
transformations and, 747–757
updates and, 784–785
query processing, 689–734
adaptive, 786–787
aggregation and, 723
basic steps of, 689, 690
Big Data and, 470–472
blockchain databases and, 1254, 1275–1276
comparisons and, 698–699
cost analysis of, 692–695, 697, 702–704, 710–712, 715–717
CPU speeds and, 692
DDL interpreter in, 20
defined, 689
distributed databases and, 1076–1086
DML compiler in, 20
duplicate elimination and, 719–720
evaluation of expressions,

Telegram: @uni_k

www.konkur.in

file scans and, 695–697, 704–707, 727
hashing and, 712–718, 1063
history of, 26–27
identifiers and, 700
indices and, 695–697
join operation and, 704–719, 722–723, 1081–1082
materialization and, 724–725
in memory, 731–734
operation evaluation and, 690–691
parsing and translation in, 689–690
pipelining and, 691–692, 724–731
PostgreSQL and, 692, 694, 698–699
projection and, 720
recursive partitioning and, 714–715
relational algebra and, 689–691
scalability of, 471–472
selection operation and, 695–700
set operations and, 720–722
on shared-memory architecture, 1061–1064
sorting and, 701–704
SQL and, 689–690, 701, 720
syntax and, 689
query processor, 18, 20
query-server systems, 963
queueing systems, 1212–1213
queueing theory, 1213
quorum consensus protocol, 1124–1125

Raft protocol, 1148, 1155–1158, 1267
RAID. ***See*** **redundant arrays of independent disks**
random access, 567, 578
randomized retry, 1148
random merge, 1056
random samples, 761
range partitioning, 476, 1005, 1007, 1178
range-partitioning sort, 1041–1042
range-partitioning vector, 1008–1009
range queries, 638, 672, 674, 1006, 1190
ranking, 219–223, 383–386
raster data, 392
RDDs (Resilient Distributed Datasets), 496–499, 1061
RDF. ***See*** **Resource Description Framework**
RDMA (remote direct memory access), 979
RDNs (relative distinguished names), 1241–1242
reactionary standards, 1237
React Native framework, 428
read-ahead, 578
read authorization, 14
read committed, 821, 880, 1225
read cost, 1127–1128
read one, write all copies protocol, 1125
read one, write all protocol, 1125
read only queries, 1039
read-only transactions, 871
read phase, 866
read quorum, 1124
read uncommitted, 821
read-write contention, 1224–1225
ready state, 989, 1102
real, double precision, 67
real-time transaction systems, 894
rebuild performance, 576
recall, 386
RecLSN, 942–945
reconciliation of updates, 1142–1143
reconfiguration, 1128
record-based models, 8
record relocation, 652–653
recovery manager, 21
recovery systems, 907–948
actions following crashes, 923–925
algorithms for, 922–925, 944–946, 1276
ARIES, 941–947, 1276
atomicity and, 803, 912–922
buffer management and, 926–930
checkpoints and, 920–922, 930
commit protocols and, 1105
concurrency control and, 916
data access and, 910–912
database modification and, 915–916
distributed databases and, 1105
early lock release and, 935–941
fail-stop assumption and, 908, 1267
failure and, 907–909, 930–932
force/no-force policy and, 927
logical undo operations and, 935–941
log records and, 913–919, 926–927
log sequence number and, 941–946
main-memory databases and, 947–948
redo operation and, 915–919, 922–925
remote backup, 909, 931–935
rollback and, 916–919, 922
shadow-copy scheme and, 914
snapshot isolation and, 916
steal/no-steal policy and, 927
storage and, 908–912, 920–922, 930–931
successful completion and, 909
transactions and, 21, 803, 805
triggers and, 212–213
undo operation and, 915–919, 922–925
write-ahead logging rule and, 926–929
recovery time, 933
recursive partitioning, 714–715
recursive queries, 213–218
iteration and, 214–216
SQL and, 216–218
transitive closure and,

Telegram: @uni_k

www.konkur.in

recursive relationship sets, 247–248
Redis, 436–437, 482, 1024
redistribution of pointers, 646
redo-only log records, 918
redo operation, 915–919, 922–925, 941
redo pass, 944–945
redo phase, 923, 924
reduceByKey function, 498
reduce function, 483–494, 510. *See also* **MapReduce** paradigm
reduce key, 485
redundancy
in database design, 243
entity-relationship (E-R) model and, 261–264
in file-processing systems, 6
reliability improvement via, 570–571
of schemas, 269–270
redundant arrays of independent disks (RAID), 570–577
bit-level striping and, 571–572
block-level striping and, 572
hardware issues, 574–576
hot swapping and, 575
levels, 572–574, 572n4, 573n6, 576–577
mirroring and, 571–573, 576, 577
parity bits and, 572, 574, 577
performance improvement via parallelism, 571–572
performance tuning and, 1214, 1229–1230
purpose of, 562
rebuild performance of, 576
recovery systems and, 909
reliability improvement via redundancy, 570–571
scrubbing and, 575
software RAID, 574, 575
striping data and, 571–572
re-engineering, 1236
referenced relation, 45–46
references, 149–150
references privilege, 170
reference types, 380–381
referencing new row as clause, 207, 208
referencing new table as clause, 210
referencing old row as clause, 208
referencing old table as clause, 210
referencing relation, 45–46
referential integrity, 14, 46, 149–153, 207–208, 800
referer, 440
referrals, 1243
ref from clause, 380
reflexivity rule, 321
region quadtrees, 1187
region queries, 393–394
regression, 546, 1234
reification, 376
reintegration, 1128
relation, defined, 39
relational algebra, 48–58
antijoin operation, 108
assignment operation, 55–56
Big Data and, 494–500
Cartesian-product operation, 50–52
equivalence and, 58, 747–757
expression transformation and, 747–757
join operation, 52–53
motivation for, 495–496
multiset, 80, 97, 108, 136, 747
predefined functions supported by, 48n4
project operation, 49–50
query optimization and, 743–749, 752, 755
query processing and, 689–691
rename operation, 56–57
select operation, 49
semijoin operation, 108, 1082–1084
set operations, 53–55
in Spark, 495–500, 508
SQL operations and, 80
on streams, 504, 506–508
unary vs. binary operations in, 48
relational-algebra expressions, 50
relational database design, 303–351
atomic domains and, 342–343
Boyce–Codd normal form and, 313–316
closure of a set and, 312, 320–324
decomposition and, 305–313, 330–341
design process, 343–347
features of good designs, 303–308
first normal form and, 342–343
fourth normal form and, 336, 339–341
functional dependencies and, 308–313, 320–330
larger schemas and, 330, 346
multivalued dependencies and, 336–341
naming of attributes and relationships in, 345–346
normalization and, 308
second normal form and, 316n8, 341–342, 356
smaller schemas and, 305, 308, 344
temporal data modeling and, 347–351
third normal form and, 317–319
relational model, 37–59
conceptual-design process for, 17
disadvantages of, 26
history of, 26
keys for, 43–46
object-relational, 27, 376–382
operations in, 48–58
overview, 8, 37
query languages in, 47–48
schema diagrams for, 46–47
schema in (*see* relational schema)
SQL language in, 13
structure of, 37–40
tables in, 9, 10, 37–40

Telegram: @uni_k

www.konkur.in

relational OLAP (ROLAP), 535
relational schema
Boyce–Codd normal form and, 313–316, 330–333
canonical cover and, 324–328
database design process and, 343–347
decomposition and, 305–313
defined, 41
in first normal form, 342–343
fourth normal form and, 339–341
functional dependencies and, 308–313, 320–330
horizontal partitioning of, 1216–1217
logically implied, 320
multivalued dependencies and, 336–341
reduction of entity-relationship diagrams to, 264–271, 277–279
redundancy in, 243
temporal data and, 347–351
third normal form and, 317–319, 333–335
for university databases, 41–43, 303–305
relation instance, 39, 41
relation scans, 769
relationship instance, 246–247
relationships, defined, 246
relationship sets
alternative notations for, 285–291
atomic domains and, 342–343
binary, 249, 283–285
combination of schemas and, 270–271
degree of, 249
descriptive attributes and, 248
design issues and, 282–285
entity-relationship diagrams and, 247–250, 268–271
entity-relationship (E-R) model and, 246–249, 282–285
mapping cardinalities and, 252–256
naming of, 345–346
nonbinary, 283–285
participation in, 247
primary keys and, 257–259
recursive, 247–248
redundancy and, 269–270
representation of, 268–269
ternary, 249, 250, 284
Unified Modeling Language and, 288–291
relations (tables), 8
relative distinguished names (RDNs), 1241–1242
relevance
hyperlinks and, 385–386
PageRank and, 385–386
TF-IDF approach and, 384–385
relevance ranking, 383–386
reliability, improvement via redundancy, 570–571
remapping bad sectors, 565
remote backup systems, 909, 931–935
remote direct memory access (RDMA), 979
rename operation, 56–57, 79, 81–82
renewing leases, 1115
reorganization of files, 598
repeatable read, 821
repeating history, 924
repeat loop, 214–216, 323
repeat statements, 201
replication
asynchronous, 1122, 1135–1138
biased protocol and, 1124
Big Data and, 481–482
caching, 1014n4
chain replication protocol, 1127–1128
concurrency control and, 1123–1125
consistency and, 1015–1016, 1121–1123, 1133–1146
data centers and, 1014–1015
distributed databases and, 987, 1121–1128
failure and, 1125–1128
key-value storage systems and, 476
location of, 1014–1015
majority protocol and, 1123–1126
master replica, 1016
master-slave, 1137
multimaster, 1137
parallel databases and, 1013–1016
primary copy, 1123
quorum consensus protocol and, 1124–1125
reconfiguration and reintegration, 1128
sharding and, 476
state machines and, 1158–1161
synchronous, 522–523, 1136
two-phase commit protocol and, 1016
update-anywhere, 1137
updates and, 1015–1016
view maintenance and, 1138–1140
report generators, 538–540
Representation State Transfer (REST), 426–427
request forgery, 439–440
request operation
deadlock handling and, 849–853
locks and, 835–841, 844–846, 849–853, 886
multiple granularity and, 854, 855
multiversion schemes and, 871
snapshot isolation and, 874
timestamps and, 861
Resilient Distributed Datasets (RDDs), 496–499, 1061
resolution of conflicting updates, 1142–1143
resource consumption, 694–695, 1065–1066
Resource Description Framework (RDF)

Telegram: @uni_k

www.konkur.in

graph representation of, 374–375
n-ary relationships and, 376
overview, 368
reification in, 376
SPARQL and, 375–376
triple representation and, 372–374
resources, in RDF, 372
response time
application design and, 434–435, 1229, 1232
blockchain databases and, 1274
parallel databases and, 971–972
partitioning and, 1007
query-evaluation plans and, 694–695
query processing and, 694–695
skew and, 1066
storage and, 572
transactions and, 808–809
response time cost model, 1066
responsive user interfaces, 423
REST (Representation State Transfer), 426–427
restriction, 172, 328, 339–340
ResultSet object, 185, 187–188, 191–193, 638–639
resynchronization, 575
reverse engineering, 1236
revoke privileges, 166–167, 171–173
right outer join, 132–136, 722
rigorous two-phase locking protocol, 842, 843
Rijndael algorithm, 448
ring system, 976
robustness, 1121, 1125–1126
ROLAP (relational OLAP), 535
roles
authorization and, 167–169
entity, 247–248
indicators associated with, 268
Unified Modeling Language and, 289
rollback
ARIES and, 945–946
cascading, 820–821, 841–842
concurrency control and, 841–844, 849–850, 853, 868–871
logical operations and, 937–939
partial, 853
recovery systems and, 916–919, 922
remote backup systems and, 933
timestamps and, 862–865
total, 853
transactions and, 143–145, 193, 196, 805–806, 922, 937–939, 945–946
undo operation and, 916–919, 937–939
rollback work, 143–145
rolling merge, 1178
rollup clause, 228
rollup construct, 227–231, 530–531, 536–538
rotational latency time, 566
round-robin scheme, 1005, 1006
routers, 1012–1013
row-level authorization, 173
row-oriented storage, 611, 615
row stores, 612, 615
R-timestamp, 862, 865, 870
R-trees, 663, 670, 674–676, 1187–1190
Ruby on Rails, 417, 419
rules for data mining, 540
runs, 701–702
runstats, 761

SAML (Security Assertion Markup Language), 442–443
SAN. *See* storage area network
sandbox, 205
SAP HANA, 615, 1132
SAS (Serial Attached SCSI) interface, 562
SATA (Serial ATA) interface, 562, 568, 569
savepoints, 947
scalability, 471–472, 477, 482, 1276
scalar subqueries, 106–107
scaleup, 972–974
scanning a relation, 1006
scheduling
disk-arm, 578–579
query optimization and, 1065
transactions and, 810–811, 811n1
schema diagrams, 46–47
schemas
alternative notations for modeling data, 285–291
authorization on, 170
basic SQL query structures and, 71–79
combination of, 270–271
composite attributes in, 250
concurrency control and (*see* concurrency control)
creation of, 24
data mining, 540–549
data warehousing, 523–525
defined, 12
entity-relationship diagrams and, 264–271, 277–279
entity-relationship (E-R) model and, 244, 246, 269–270, 277–279
evolution of, 292
flexibility of, 366
global, 1076, 1078–1079
integration of, 1076, 1078–1080
intermediate SQL and, 162–163
larger, 330, 346
local, 1076
locks and (*see* locks)
logical, 12–13, 1223–1224
performance tuning of, 1214–1218, 1223–1224
physical, 12, 13
physical-organization modification of, 25
P + Q redundancy, 573, 574
recovery systems and (*see* recovery systems)
redundancy of, 269–270
relational (*see* relational

Telegram: @uni_k

www.konkur.in

relationship sets and, 268–269
shadow-copy, 914
smaller, 305, 308, 344
snowflake, 524
SQL DDL and, 24, 66, 68–71
star, 524–525
strong entity sets and, 265–267
subschemas, 12
timestamps and, 861–866
for university databases, 1287–1288
version-numbering, 1141
version-vector, 1141–1142
weak entity sets and, 267–268
SciDB, 367
SCM (storage class memory), 569, 588, 948
SCOPE, 494
scope clause, 380
scripting languages, 404–405, 416–418, 421, 439
scrubbing, 575
SCSI (small-computer-system interconnect), 562, 563
search keys
B^+-trees and, 634–650
hashing and, 1190–1195, 1197–1199
index creation and, 664
nonunique, 632, 637, 640, 649–650
ordered indices and, 624–634
storage and, 595, 597–598
uniquifiers and, 649–650
search operation, 1188
secondary indices, 625, 632–633, 652–653, 695–698, 1017–1019
secondary site, 931
secondary storage, 561
second normal form (2NF), 316n8, 341–342, 356
sectors, 564–566
security
abstraction levels and, 12
application design and, 437–446
audit trails and, 445–446
authentication (*see* authentication)
authorization (*see* authorization)
of blockchain databases, 1253–1255, 1259
concurrency control and (*see* concurrency control)
cross-site scripting and, 439–440
dictionary attacks and, 449
encryption and, 447–453
end-user information and, 443
in file-processing systems, 7
GET method and, 440
integrity manager and, 19
locks and (*see* locks)
man-in-the-middle attacks and, 442
passwords (*see* passwords)
privacy and, 438, 446
request forgery and, 439–440
single sign-on system and, 442–443
SQL DDL and, 67
SQL injection and, 438–439
unique identification and, 446, 446n8
Security Assertion Markup Language (SAML), 442–443
seek time, 566, 566n2, 567, 692, 710
select all, 73
select authorization, privileges and, 166–167
select clause
aggregate functions and, 91–96
attribute specification in, 83
basic SQL queries and, 71–79
on multiple relations, 74–79
in multiset relational algebra, 97
null values and, 90
OLAP and, 534, 536–537
ranking and, 220–222
rename operation and, 79, 81–82
set membership and, 98–99
set operations and, 85–89
on single relation, 71–74
string operations and, 82–83
select distinct, 72–73, 90, 99–100, 142
select-from-where
deletion and, 108–110
function/procedure writing and, 199–206
insertion and, 110–111
join expressions and, 125–136
natural joins and, 126–130
nested subqueries and, 98–107
updates and, 111–114
views and, 137–143
selection
comparisons and, 698–699
complex, 699–700
conjunctive, 699–700, 747, 762
disjunctive, 699, 700, 762
equivalence and, 747–757
file scans and, 695–697, 704–707, 727
identifiers and, 700
indices and, 695–697, 783
intraoperation parallelism and, 1049
linear search and, 695
size estimation and, 760, 762
view maintenance and, 780–781
selectivity, 762
select operation, 49
select privilege, 171, 172
semijoin operation, 108, 776, 1082–1084
semi-structured data models, 365–376
flexible schema and, 366
history of, 8, 27
JSON, 8, 27, 367–370
knowledge representation and, 368
motivations for use of, 365–366
multivalued, 366–367

Telegram: @uni_k

www.konkur.in

overview, 366–368
RDF and knowledge graphs, 368, 372–376
XML, 8, 27, 367–368, 370–372
sensor data, 470, 501
sentiment analysis, 549
Sequel, 65
sequence counters, 1226
sequential-access storage, 561, 567, 578
sequential computation, 974
sequential file organization, 595, 597–598
Serial ATA (SATA) interface, 562, 568, 569
Serial Attached SCSI (SAS) interface, 562
serializability
blind writes and, 868
concurrency control and, 836, 840–843, 846–848, 856, 861–871, 875–887
conflict, 813–816
isolation and, 821–826
operation, 885
order of, 817
performance tuning and, 1225
precedence graph and, 816–817
in the real world, 824
snapshot isolation and, 875–879
topological sorting and, 817–818
transactions and, 812–819, 821–826
view, 818–819, 867–868
serializable schedules, 811, 812
serializable snapshot isolation (SSI) protocol, 878
server-side scripting, 416–418
server systems, 962–970
categorization of, 963–964
data-server, 963–964, 968–970
defined, 962
transaction-server, 963–968
tree-like, 977–978
services, 994
service time, 1230
servlets, 411–421
alternative server-side frameworks, 416–421
example of, 411–413
life cycle and, 415–416
server-side scripting and, 416–418
sessions and, 413–415
support for, 416
web application frameworks and, 418–419
session window, 505
set autocommit off, 144
set clause, 113
set default, 150
set-difference operation, 55, 750
set null, 150
set operations
except, 88–89
intersect, 54–55, 87–88, 750
nested subqueries and, 98–101
query optimization and, 764–765
query processing and, 720–722
set comparison and, 99–101
set difference, 55
union, 53–54, 86–87, 750
set orientation, 1220–1221
set role, 172
set statement, 201, 209
set transaction isolation level serializable, 822
set types, 366, 367
shadow-copy scheme, 914
shadowing, 571
shadow paging, 914
SHA-256 hash function, 1260
sharding, 473, 475–476, 1275
shard key, 479
shared and intention-exclusive (SIX) mode, 855
shared-disk architecture, 979, 980, 984–985
shared locks, 825, 835, 854, 880, 1124
shared-memory architecture, 21–22, 979–984, 1061–1064
shared-mode locks, 835
shared-nothing architecture, 979, 980, 985–986, 1040–1041, 1061–1063
shared scans, 785–786
Sherpa/PNUTS, 477, 1024, 1026, 1028–1030
show warnings, 746
shrinking phase, 841, 843
shuffle step, 486, 1061
SIMD (Single Instruction Multiple Data), 1064
simple attributes, 250, 265
simulation model, 1230
single inheritance, 275
Single Instruction Multiple Data (SIMD), 1064
single lock-manager, 1111
single sign-on system, 442–443
single-user systems, 962
single-valued attributes, 251
sites, 986
SIX (shared and intention-exclusive) mode, 855
skew
aggregation and, 1049–1050
attribute-value, 1008
data distribution, 1008
in distribution of records, 660
execution, 1007, 1008, 1043
hash indices and, 1194
in joins, 1047–1048
parallel databases and, 974, 1007–1013, 1043, 1062
partitioning and, 715, 1007–1013
response time and, 1066
write skew, 876–877
slicing, 530
sliding window, 505
slotted-page structure, 593
small-computer-system interconnect (SCSI), 562, 563

Telegram: @uni_k

www.konkur.in

smart contracts, 1258, 1269–1273
snapshot isolation
concurrency control and, 872–879, 882, 916, 1131–1132
distributed, 1131–1132
performance tuning and, 1225
recovery systems and, 916
serializability and, 875–879
transactions and, 825–826, 1136
validation and, 874–875
snowflake schema, 524
social-networking sites
Big Data and, 467, 470
database applications for, 2, 3, 27
key-value store and, 471
streaming data and, 502
soft deadlines, 894
soft forks, 1257, 1258
software-as-a-service model, 993
software RAID, 574, 575
solid-state drives (SSDs), 18, 560, 568–570, 693n3, 1229
some construct, 100, 100n10
some function, 96
sophisticated users, 24
sorting
cost analysis of, 702–704
duplicate elimination and, 719–720
external sort-merge algorithm, 701–704
parallel external sort-merge, 1042–1043
query processing and, 701–704
range-partitioning, 1041–1042
topological, 817–818
sort-merge-join algorithm, 708–712
sound axioms, 321
source-driven architecture, 522
space overhead, 624, 627–628, 634, 1202
Spark, 495–500, 508, 511, 1061
SPARQL, 375–376
sparse column data representation, 366
sparse indices, 626–632
spatial data
design databases, 390–391
geographic, 387, 390–393
geometric, 388–390
indexing of, 672–675, 1186–1190
joins over, 719
quadtrees, 392, 674, 1186–1187
queries and, 393–394
R-trees, 663, 670, 674–676, 1187–1190
triangulation and, 388, 393
spatial data indices, 672–675
spatial graph queries, 394
spatial graphs, 390
spatial-integrity constraints, 391
spatial join, 394
spatial networks, 390
specialization
attribute inheritance and, 274–275
constraints on, 275–276
disjoint, 272, 275
entity-relationship (E-R) model and, 271–273
overlapping, 272, 275
partial, 275
single entity set and, 274
superclass-subclass relationship and, 272
total, 275
specification of functional requirements, 17–18, 242
speedup, 972–974
splitting nodes, 641–645, 886
SQL Access Group, 1238
SQLAlchemy, 382
SQL/DS, 26
SQL environment, 163
SQL injection, 189, 438–439
SQLite, 668
SQLLoader, 1222
SQL MED, 1077
sql security invoker, 170
sqlstate, 205
SQL (Structured Query Language), 65–114
advanced (*see* advanced SQL)
aggregate functions and, 91–96
application-level authorization and, 443–445
application programs and, 16–17
attribute specification in select clause, 83
authorization and, 66
basic types supported by, 67–68
blobs and, 156, 193, 594, 652
bulk loads and, 1221–1223
clobs and, 156, 193, 594, 652
create table and, 68–71
database modification and, 108–114
data definition for university databases, 69–71, 1288–1292
DDL and, 14–15, 65–71
decision-support systems and, 521
deletion and, 108–110
DML and, 16, 66
dumping and, 931
dynamic, 66, 184, 201
embedded, 66, 184, 197–198, 965, 1269
index creation and, 164–165, 664–665
injection and, 189, 438–439
inputs and outputs in, 747
insertion and, 110–111
integrity constraints and, 14–15, 66, 145–153
intermediate (*see* intermediate SQL)
isolation levels and, 821–826
JSON and, 369–370
limitations of, 468, 472
on MapReduce, 493–494
MySQL (*see* MySQL)
nested subqueries and, 98–107

Telegram: @uni_k

www.konkur.in

NoSQL systems, 28, 473, 477, 1269, 1276
null values and, 89–90
OLAP in, 533–534, 536–538
ordering display of tuples and, 83–84
overview, 65–66
PostgreSQL (*see* PostgreSQL)
prepared statements and, 188–190
prevalence of use, 13
query processing and, 689–690, 701, 720
query structure, 71–79
relational algebra and, 80
rename operation and, 79, 81–82
ResultSet object and, 185, 187–188, 191–193, 638–639
schemas and, 24, 66, 68–71
security and, 438–439
set operations and, 85–89
standards for, 65, 1237–1238
stream extensions to, 504–506
string operations and, 82–83
System R and, 26
System R project and, 65
theoretical basis of, 48
transactions and, 66, 143–145, 965
updates and, 111–114
views and, 66, 137–143, 169–170
where-clause predicates and, 84–85
XML and, 372
SSDs. *See* solid-state drives
SSI (serializable snapshot isolation) protocol, 878
stable storage, 804–805, 908–910
stale messages, 1149
stalls in processing, 733
standards
ANSI, 65, 1237
anticipatory, 1237
CLI, 197, 1238–1239
database connectivity, 1238–1239
de facto, 1237
defined, 1237
formal, 1237
ISO, 65, 1237, 1241
object-oriented, 1239–1240
ODBC, 1238–1239
reactionary, 1237
SQL, 65, 1237–1238
X/Open XA, 1239
star schema, 524–525
start-up costs, 974, 1066
start with/connect by prior syntax, 218
starved transactions, 841, 853
state-based blockchains, 1269, 1271
state machines, 1158–1161
Statement object, 186–187, 189
state of execution, 727
static hashing, 661, 1190–1195, 1202–1203
statistical analysis, 520, 527
statistics, 757–766
catalog information and, 758–760
computing, 761
join size estimation and, 762–764
maintaining, 761
number of distinct values and, 765–766
selection size estimation and, 760, 762
steal policy, 927
stepped-merge indices, 667, 1179–1181
stock market, streaming data and, 501
stop words, 385
storage, 587–617
access time and, 561, 566, 567, 578
architecture for, 587–588
archival, 561
atomicity and, 804–805
authorization and, 19
backup (*see* backup)
Big Data and, bit-level striping and, 571–572
blockchain (*see* blockchain databases)
block-level striping and, 572
buffers and, 19, 604–610
byte amounts of, 18
checkpoints and, 920–922, 930
cloud-based, 28, 563, 992–993
column-oriented, 525–526, 588, 611–617, 734, 1182
crashes and, 607, 609–610
data access and, 910–912
data-dictionary, 602–604
data mining and, 27, 540–549
data-transfer rate and, 566, 569
in decision-support systems, 519–520
direct-access, 561
distributed (*see* distributed databases)
distributed file systems for, 472–475, 489, 1003, 1019–1022
dumping and, 930–931
durability and, 804–805
elasticity of, 1010
file manager for, 19
file organization and, 588–602
force output and, 912
geographically distributed, 1026–1027
hard disks for, 26
integrity manager and, 19
key-value, 471, 473, 476–480, 1003, 1023–1031
of large objects, 594–595
log disks, 610
in main-memory databases, 588, 615–617
mirroring and, 571–573, 576, 577
non-volatile, 560, 562, 587–588, 804, 908–910, 930–931
offline, 561

Telegram: @uni_k

www.konkur.in

outsourcing, 28
parallel (*see* parallel databases)
physical (*see* physical storage systems)
pointers and, 588, 591, 594–598, 601
primary, 561
punched cards for, 25
random access, 567, 578
recovery systems and, 908–912, 920–922, 930–931
redundant arrays of independent disks, 562
response time and, 572
row-oriented, 611, 615
R-trees and, 1189–1190
scrubbing and, 575
secondary, 561
seek time and, 566, 566n2, 567, 692, 710
sequential access, 561, 567, 578
sharding and, 473, 475–476
SQL DDL and, 67
stable, 804–805, 908–910
striping data and, 571–572
structure and access-method definition, 24
tertiary, 561
transaction manager for, 19
volatile, 560, 562, 804, 908
wallets and, 450
warehousing (*see* data warehousing)

storage area network (SAN), 562, 563, 570, 934, 985
storage class memory (SCM), 569, 588, 948
storage manager, 18–20
store barrier, 983
stored functions/procedures, 1031
straggler nodes, 1060–1061
streaming data, 500–508
algebraic operations and, 504, 506–508
applications of, 500–502
continuous, 731
defined, 500
fault tolerance with, 1074–1076
processing, 468, 1070–1076
querying, 502–506, 1070–1071
routing of tuples and, 1071–1073

stream query languages, 503–506
strict two-phase locking protocol, 842, 843
string operations
aggregate, 91
escape, 83
JDBC and, 184–193
like, 82–83
lower function, 82
query result retrieval and, 188
similar to, 83
trim, 82
upper function, 82

stripe, 613–614
striping data, 571–572
strong entity sets, 259, 265–267
Structured Query Language. ***See*** **SQL**
structured types, 158–160
stylesheets, 408
subject of triples, 372
sublinear scaleup, 973
sublinear speedup, 972
subschemas, 12
suffix, 1243
sum function, 91, 139, 228, 536, 723, 766, 781
superclass-subclass relationship, 272, 274
superkeys, 43–44, 257–258, 309–310, 312
supersteps, 510
superusers, 166
supply chains, 1266, 1278
support, 547
Support Vector Machine (SVM), 544–545
swap space, 929
Sybase IQ, 615
Sybil attacks, 1255, 1256, 1264, 1266
symmetric fragment-and-replicate joins, 1046
symmetric-key encryption, 448
synchronous replication, 522–523, 1136
syntax, 199, 201–205, 689
sys_context function, 173
system architecture. ***See*** **architecture**
system catalogs, 602–604, 1009
system clock, 862
system error, 907
System R, 26, 65, 772–773, 772n3

table alias, 81
table functions, 200
table inheritance, 379–380
table partitioning, 601–602
tables
defined, 1011
dimension, 524
dirty page, 941–947
distributed hash, 1013
fact, 524
foreign, 1077
partition, 1011–1014
pivot-table, 226–227, 528–529
in relational model, 9, 10, 37–40
in SQL DDL, 14–15
transition, 210

tablets, 1011, 1025
tablet server, 1025
tab-separated values, 1222
tag library, 418
tags, 370–372, 406–407, 418, 440
tamper resistance, 1253–1255, 1259, 1260
tangles, 1278
tape storage, 561
Tapestry, 419
tasks, 1051–1052. ***See also*** **workflow**
Tcl, 206
telecommunications, database applications for, 3

Telegram: @uni_k

www.konkur.in

temporal data indices, 675–676
temporal validity, 157
term frequency (TF), 384
termination of transactions, 806
terms, 384, 1149
ternary relationship sets, 249, 250, 284
tertiary storage, 561
test-and-set, 966
test suites, 1234–1235
text mining, 549
textual data, 382–387. ***See also*** **information retrieval**
keyword queries, 383, 386–387
overview, 382
relevance ranking and, 383–386
Tez, 495
TF-IDF approach, 384–385
TF (term frequency), 384
then clause, 212
theta-join operations, 748–749
third normal form (3NF), 317–319, 333–335
Thomas' write rule, 864–866
threads, 965, 982, 1062
3D-XPoint memory technology, 569
3NF decomposition algorithm, 334–335
3NF synthesis algorithm, 335
3NF (third normal form), 317–319, 333–335
three-phase commit (3PC) protocol, 1107
three-tier architecture, 23
throughput
application design and, 1230–1232, 1234
in blockchain databases, 1274
harmonic mean of, 1231
improved, 808
parallel databases and, 971
range partitioning and, 1007
storage and, 572
system architectures and, 963, 971
transactions and, 808
throughput test, 1234
tickets and ticketing, 1133, 1279
tiles, 392
time intervals, 675–676
time-lock transactions, 1273
timestamps
concurrency control and, 861–866, 882
for data-storage systems, 480
defined, 154
distributed databases and, 1116–1118
generation of, 1117–1118
invalidation, 873n3
logical clock, 1118
logical counter and, 862
multiversion schemes and, 870–871
nondeterministic, 508n3
ordering scheme and, 862–864, 870–871, 1118
rollback and, 862–865
snapshot isolation and, 873, 873n2
system clock and, 862
Thomas' write rule and, 864–866
transactions and, 825
tuples and, 502, 503, 505
time to completion, 1231
timezone, 154
TIN (triangulated irregular network), 393
tokens, 1272
Tomcat Server, 416
top-down design, 273
topic-partition system, 1073
top-*K* optimization, 784
topographical information, 393
topological sorting, 817–818
toss-immediate strategy, 608
total failure, 909
total generalization, 275
total rollback, 853
total specialization, 275
TPC (Transaction Processing Performance Council), 1232–1234
TPS (transactions per second), 1232
tracks, 564
training instances, 541, 543, 546
transaction control, 66
transaction coordinators, 1099, 1104, 1106–1107
transaction identifiers, 913
transaction managers, 18–21, 1098–1099
Transaction Processing Performance Council (TPC), 1232–1234
transactions, 799–828
aborted, 805–807, 819–820
actions following crashes, 923–925
active, 806
aggregation of, 1278
alternative models of processing, 1108–1110
association rules and, 546–547
atomicity of, 20–21, 144, 481, 800–807, 819–821
begin/end operations and, 799
blockchain, 1261–1263, 1268–1271, 1273
cascadeless schedules and, 820–821
check constraints and, 800
commit protocols and, 1100–1110
committed (*see* committed transactions)
commit work and, 143–145
compensating, 805
concept of, 799–801
concurrency control and (*see* concurrency control)
concurrent, 1224–1227
consistency of, 20, 800, 802, 807–808, 821–823
crashes and, 800
cross-chain, 1273
data mining, 540–549
defined, 20, 799
distributed, 989–990, 1098–1100
double-spend, 1261–1262, 1264

Telegram: @uni_k

www.konkur.in

failure of, 806, 907, 909, 1100
force/no-force policy and, 927
gas concept for, 1270–1271
global, 988, 1098, 1132
in-doubt, 1105
integrity constraint violation and, 151–152
isolation of, 800–804, 807–812, 819–826
killing, 807
local, 988, 1098, 1132
locks and (*see* locks)
log of (*see* log records)
long-duration, 890–891
minibatch, 1227
multiversion schemes and, 869–872, 1129–1131
observable external writes and, 807
off-chain, 1275
online, 4, 521
performance tuning of, 1224–1227
persistent messaging and, 990, 1016, 1108–1110, 1137
read-only, 871
real-time systems, 894
recoverable schedules and, 819–820
recovery systems and, 21, 803, 805
remote backup systems and, 931–935
restarting, 807
rollback and, 143–145, 193, 196, 805–806, 922, 937–939, 945–946
scalability and, 471
serializability and, 812–819, 821–826
shadow-copy scheme and, 914
simple model for, 801–804
as SQL statements, 826–828
starved, 841, 853
states of, 805–807
steal/no-steal policy and, 927
storage structure and, 804–805
support for, 1028–1030
terminated, 806
time-lock, 1273
timestamps and, 861–866
two-phase commit protocol and, 989, 1016, 1276
as unit of program execution, 799
update, 871
validation and, 866–869
wait-for graph and, 851–852, 1113–1114
write-ahead logging rule and, 926–929
write operations and, 826
transaction scaleup, 973
transactions-consistent snapshot, 1136
transaction-server systems, 963–968
transactions per second (TPS), 1232
transaction time, 347n10
TransactSQL, 199
transfer of control, 932–933
transformations
data warehousing and, 523
equivalence rules and, 747–752
examples of, 752–754
join ordering and, 754–755
query optimization and, 747–757
relational algebra and, 747–757
transition tables, 210
transition variables, 207
transitive closure, 214–216
transitive dependencies, 317n9, 356
transitivity rule, 321
translation, query processing and, 689–690
translation table, 568
tree-like server systems, 977–978
tree-like topology, 977
tree protocol, 846–848
trees
B (*see* B-trees)
B^+ (*see* B^+-trees)
B-link, 886
decision-tree classifiers, 542
directory information, 1242, 1243
disjoint subtrees, 847
Generalized Search Tree, 670
k-d, 673–674
k-d B, 674
left-deep join, 773
LSM, 666–668, 1028, 1176–1182, 1215
Merkle, 1143–1146, 1268, 1269
Merkle-Patricia, 1269, 1275
multiple granularity and, 853–857
operator, 724, 1040
quadratic split heuristic and, 1189
quadtrees, 392, 674, 1186–1187
R, 663, 670, 674–676, 1187–1190
tree topology, 977
triangulated irregular network (TIN), 393
triangulation, 388, 393
triggers
alter, 210
defined, 206
disable, 210
drop, 210
need for, 206–207
nonstandard syntax and, 212
recovery and, 212–213
in SQL, 207–210
transition tables and, 210
when not to use, 210–213
trim, 82
triple representation, 372–374
trivial functional dependencies, 311
true predicate, 76
true values, 96, 101
try-with-resources construct, 187, 187n3
tumbling window, 505
tuning. *See* performance tuning
tuning wizards, 1215
tuple-generating dependencies, 337

Telegram: @uni_k

www.konkur.in

aggregate functions and, 91–96
in Cartesian-product operation, 51, 52
defined, 39
deletion and, 108–110, 613
duplicate, 103
eager generation of, 726, 727
insertion and, 110–111
join operation for, 52–53 (*see also* joins)
lazy generation of, 727
logical routing of, 506–507, 1071–1073
ordering display of, 83–84
physical routing of, 1072–1073
pipelining and, 691–692, 724–731
query optimization and (*see* query optimization)
query processing and (*see* query processing)
ranking and, 219–223
reconstruction costs, 612–613
relational algebra and, 747–757
in relational model, 39, 41, 43–46
select operation for, 49
set operations and, 54–55, 85–89
streaming data and, 501–503, 505–507, 1071–1073
timestamps and, 502, 503, 505
updates and, 111–114, 613
views and, 137–143
windowing and, 223–226
tuple variables, 81
Turing-complete languages, 1258, 1269, 1270
two-factor authentication, 441–442
2NF (second normal form), 316n8, 341–342, 356
two-phase commit (2PC) protocol, 989, 1016, 1101–1107, 1161, 1276
two-phase locking protocol, 841–844, 871–872, 1129–1131
two-tier architecture, 23
type inheritance, 378–379
types
blobs, 156, 193, 594, 652
clobs, 156, 193, 594, 652
complex (*see* complex data types)
large-object, 156, 158
performance tuning and, 1226
reference, 380–381
user-defined, 158–160, 378

UML (Unified Modeling Language), 288–291
unary operations, 48
undo operation
concurrency control and, 940–941
logical, 936–941
recovery systems and, 915–919, 922–925
rollback and, 916–919, 937–939
undo pass, 944–946
undo phase, 923–925
Unified Modeling Language (UML), 288–291
uniform resource locators (URLs), 405–406
union all, 86, 97, 217n8
union of sets, 54, 750
union operation, 53–54, 86–87, 228
union rule, 321
unique construct, 103, 147
unique key values, 160–161
unique-role assumption, 345
uniquifiers, 649–650
United States
address format used in, 250n1
identification numbers in, 447
primary keys in, 44–45
privacy laws in, 995
universal front end, 404
Universal Serial Bus (USB) slots, 560
universal Turing

university databases
abstraction levels for, 11–12
application design and, 2–3, 5–7, 403–404, 431, 442–444
atomic domains and, 343
Big Data and, 499–500
blockchain, 1277
buffer-replacement strategies and, 607–609
Cartesian product and, 76–79
combination of schemas and, 270–271
complex attributes and, 249–252
concurrency control and, 879
consistency constraints for, 6, 13–15
decomposition and, 305–307, 310–312
deletion requests and, 109–110
design issues for, 346–347
distributed, 988
entities in, 243–246, 265–268, 281–283
entity-relationship diagram for, 263–264
full schema for, 1287–1288
functions and procedures for, 198–199
generalization and, 273–274
hash functions and, 1190–1193
incompleteness of, 243–244
indices and, 625–628, 664, 1017–1018
insertion requests and, 110–111
integrity constraints for, 145–153
mapping cardinalities and, 253–256
multivalued dependencies and, 336
query optimization and, 751–755, 775–778
query processing and,

Telegram: @uni_k

www.konkur.in

recursive queries and, 213–214, 217–218
redundancy in, 243, 261–264, 269–270
relational algebra for, 49–58
relational model for, 9, 10, 37–47
relational schema for, 41–43, 303–305
relationship sets and, 246–249, 268–269, 282–283
roles and authorizations for, 167–169
sample data for, 1292–1298
specialization and, 271–273
SQL data definition for, 69–71, 1288–1292
SQL queries for, 16
storage and, 589–591, 597–601
transactions and, 144, 826–827
triggers and, 211–213
triple representation of, 373–374
unique key values for, 160–161
user interfaces for, 24
views and, 141–143
University of California, Berkeley, 26
Unix, 83, 914
unknown values, 89–90, 96
unpartitioned site, 1056
unpin operations, 605
updatable result sets, 193
update-anywhere replication, 1137
update hot spots, 1225
updates
authorization and, 14, 170, 171
batch, 1221
on B^+-trees, 641–649
complexity of, 647–649
database modification and, 111–114
data warehousing and, 523
deletion time and, 641, 645–649
EXEC SQL and, 197
hashing and, 624, 1197–1202
inconsistent, 1140–1142
indices and, 630–632
insertion time and, 641–645, 647, 649
lazy propagation of, 1122, 1136
log records and, 913–914, 917–918
lost, 874
LSM trees and, 1178–1179
performance tuning and, 1221–1223, 1225–1227
privileges and, 166–167
query optimization and, 784–785
reconciliation of, 1142–1143
replication and, 1015–1016
shipping SQL statements to database and, 187
snapshot isolation and, 873–879
triggers and, 208, 212
tuples and, 111–114, 613
of views, 140–143
update transactions, 871
upgrade, 843
URLs (uniform resource locators), 405–406
U.S. National Institute for Standards and Technology, 288
USB (Universal Serial Bus) slots, 560
user-defined types, 158–160, 378
user-interface layer, 429
user interfaces
application architectures and, 429–434
application programs and, 403–405
back-end component of, 404
business-logic layer and, 430, 431
client-server architecture and, 404
client-side scripting and, 421–429
common gateway interface standard and, 409
cookies and, 410–415, 411n2, 439–440
CRUD, 419
data access layer and, 430–434
disconnected operation and, 427–428
front-end component of, 404
HTTP (*see* HyperText Transfer Protocol)
mobile application platforms and, 428–429
for naÃ¨ve users, 24
presentation layer and, 429
responsive, 423
security and, 437–446
for sophisticated users, 24
storage and, 562–563
Web services (*see* World Wide Web)
web services and, 426–429
user requirements in database design, 17–18, 241–242, 274
utilization of resources, 808

validation
concurrency control and, 866–869, 882
distributed, 1119–1120
first committer wins and, 874
first updater wins and, 874–875
phases of, 866
recovery systems and, 916
snapshot isolation and, 874–875
test for, 868
view serializability and, 867–868
validation phase, 866
valid interval, 870
valid time, 157, 347–350, 347n10
value for entity set attributes, 245
value set of attributes, 249
varchar, 67–68, 70

Telegram: @uni_k

www.konkur.in

variety of data, 468
VBScript, 417
vector data, 392–393
vector processing, 612
Vectorwise, 615
velocity of data, 468
verification of contents, 1145
version numbering, 1141
versions period for, 157
version-vector scheme, 1141–1142
Vertica, 615
vertical partitioning, 1004
view definition, 66
view equivalence, 818, 818n4
view level of abstraction, 10–12
view maintenance, 140, 779–782, 1138–1140, 1215–1216
views
authorization on, 169–170
with check option, 143
create view, 138–143, 162, 169
deferred maintenance and, 779, 1215–1216
defined, 137–138
deletion and, 142
immediate maintenance and, 779, 1215–1216
insertion and, 141–143
materialized (*see* materialized views)
performance tuning and, 1215–1216
SQL queries and, 138–139
update of, 140–143
view serializability, 818–819, 867–868
virtual machines (VMs), 970, 991–994
virtual nodes, 1009–1010
Virtual Private Database (VPD), 173, 444–445
virtual processor approach, 1010n3
Visual Basic, 184, 206
visualization tools, 538–540
VMs (virtual machines), 970, 991–994
volatile storage, 560, 562, 804, 908
volume of data, 468
VPD (Virtual Private Database), 173, 444–445

wait-die scheme, 850, 1112
wait-for graphs, 851–852, 1113–1114
WAL (write-ahead logging), 926–929, 934
WANs (wide-area networks), 989
weak entity sets, 259–260, 267–268
wear leveling, 568
web application frameworks, 418–419
web-based services, database applications for, 3
web crawlers, 383
Weblogic Application Server, 416
WebObjects, 419
web servers, 408–411
web services, 423–429
defined, 426
disconnected operation and, 427–428
interfacing with, 423–426
mobile application platforms, 428–429
web sessions, 408–411
WebSphere Application Server, 416
when clause, 212
when statement, 208
where clause
aggregate functions and, 91–96
basic SQL queries and, 71–79
between comparison, 84
on multiple relations, 74–79
in multiset relational algebra, 97
not between comparison, 84
null values and, 89–90
predicates, 84–85
query optimization and, 774–777
ranking and, 222
rename operation and, 79, 81–82
security and, 445
set operations and, 85–89
on single relation, 71–74
string operations and, 82–83
transactions and, 824, 826–827
while loop, 196
while statements, 201
wide-area networks (WANs), 989
wide column data representation, 366
wide-column stores, 1023
windows and windowing, 223–226, 502–506
WiredTiger, 1028
wireframe models, 390
with check option, 143
with clause, 105–106, 217
with data clause, 162
with grant option, 170–171
with recursive clause, 217
with timezone specification, 154
witness data, 1258
word count program, 483–486, 484n2, 490–492
workers, 1051
workflow
business-logic layer and, 431
database design and, 291–292
distributed transaction processing and, 1110
management systems for, 990
workload, 783, 1215, 1217
workload compression, 1217
work stealing, 1048, 1062
World Wide Web
application design and, 405–411
cookies and, 410–415, 411n2, 439–440
encryption and, 447–453
growth of, 467
HTML and (*see* HyperText Markup Language)
HTTP and (*see* HyperText Transfer Protocol)
impact on database systems, 27
security and, 437–446

Telegram: @uni_k

www.konkur.in

URLs and, 405–406
WORM (write once, read-many) disks, 561, 1022
wound-wait scheme, 850
wrappers, 1077, 1236
write-ahead logging (WAL), 926–929, 934
write amplification, 1180
write once, read-many (WORM) disks, 561, 1022
write operations, 826
write-optimized index structures, 665–670
write phase, 866
write quorum, 1124
write skew, 876–877
write-write contention, 1225
W-timestamp, 862, 865, 870

X.500 directory access protocol, 1241
XML (Extensible Markup Language)
emergence of, 27
flexibility of, 367–368
as semi-structured data model, 8, 27
SQL in support of, 372
tags and, 370–372
for transferring data, 423
X/Open XA standards, 1239
XOR operation, 448
XPath, 372
XQuery, 372
XSRF (cross-site request forgery), 439–440
XSS (cross-site scripting), 439–440

Yahoo Message Bus service, 1137–1138

Zab protocol, 1152
ZooKeeper, 1150, 1152

Telegram: @uni_k