DATABASE
SYSTEM CONCEPTS

SIXTH EDITION

Abraham Silberschatz
Yale University

Henry F. Korth
Lehigh University

S. Sudarshan
Indian Institute of Technology, Bombay

™ ™

DATABASE SYSTEM CONCEPTS, SIXTH EDITION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020.

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 0 DOC/DOC 1 0 9 8 7 6 5 4 3 2 1 0

ISBN 978-0-07-352332-3
MHID 0-07-352332-1

Global Publisher: *Raghothaman Srinivasan*
Director of Development: *Kristine Tibbetts*
Senior Marketing Manager: *Curt Reynolds*
Project Manager: *Melissa M. Leick*
Senior Production Supervisor: *Laura Fuller*
Design Coordinator: *Brenda A. Rolwes*
Cover Designer: *Studio Montage, St. Louis, Missouri*
(USE) Cover Image: © *Brand X Pictures/PunchStock*
Compositor: *Aptara®, Inc.*
Typeface: *10/12 Palatino*
Printer: *R. R. Donnelley*

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

Library of Congress Cataloging-in-Publication Data

Silberschatz, Abraham.
Database system concepts / Abraham Silberschatz. — 6th ed.
p. cm.
ISBN 978-0-07-352332-3 (alk. paper)
1. Database management. I. Title.
QA76.9.D3S5637 2011
005.74—dc22
2009039039

www.mhhe.com

In memory of my father Joseph
Silberschatz my mother Vera Silberschatz
and my grandparents Stepha and Aaron

Rosenblum Avi Silberschatz

To my wife, Joan
my children, Abigail and Joseph
and my parents, Henry and Frances

Hank Korth

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

S. Sudarshan

This page intentionally left blank

Contents

Chapter 4 Intermediate SQL

Chapter 5 Advanced SQL

Chapter 6 Formal Relational Query Languages

PART TWO DATABASE DESIGN

Chapter 7 Database Design and the E-R Model

Chapter 8 Relational Database Design

Chapter 9 Application Design and Development

PART THREE DATA STORAGE AND QUERYING

Chapter 10 Storage and File Structure

Chapter 11 Indexing and Hashing

Chapter 12 Query Processing

Chapter 13 Query Optimization

PART FOUR TRANSACTION MANAGEMENT

Chapter 14 Transactions

Chapter 15 Concurrency Control

Chapter 16 Recovery System

PART FIVE SYSTEM ARCHITECTURE

Chapter 17 Database-System Architectures

Chapter 18 Parallel Databases

Chapter 19 Distributed Databases

PART SIX DATA WAREHOUSING, DATA MINING, AND INFORMATION RETRIEVAL

Chapter 20 Data Warehousing and Mining

Chapter 21 Information Retrieval

PART SEVEN SPECIALTY DATABASES

Chapter 22 Object-Based Databases

Chapter 23 XML

PART EIGHT ADVANCED TOPICS

Chapter 24 Advanced Application Development

Chapter 25 Spatial and Temporal Data and Mobility

Chapter 26 Advanced Transaction Processing

PART NINE CASE STUDIES

Chapter 27 PostgreSQL

Chapter 28 Oracle

Chapter 29 IBM DB2 Universal Database

Chapter 30 Microsoft SQL Server

PART TEN APPENDICES Appendix

A Detailed University Schema

Appendix B Advanced Relational Design (contents online)

Appendix C Other Relational Query Languages (contents online)

Appendix D Network Model (contents online)

Appendix E Hierarchical Model (contents online)

This page intentionally left blank

Preface

Database management has evolved from a specialized computer application to a central component of a modern computing environment, and, as a result, knowl edge 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 lan guages, 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 organi zation, and a high-level programming language such as Java, C, or Pascal. We present concepts as intuitive descriptions, many of which are based on our run ning 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 biblio graphical 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. Details of particular database systems are discussed in Part 9, "Case Studies."

In this, the sixth 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. We have also taken into account trends in the teaching of database concepts and made adaptations to facilitate these trends where appropriate.

Organization

The text is organized in nine major parts, plus five appendices.

- **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 in terfaces 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 ex ample throughout the book. This chapter is motivational, historical, and ex planatory in nature.

- **Part 1: Relational Databases** (Chapters 2 through 6). 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, and relational operations. Chapters 3, 4, and 5 focus on the most influential of the user-oriented relational languages: SQL. Chapter 6 cov ers the formal relational query languages: relational algebra, tuple relational calculus, and domain relational calculus.

  The chapters in this part describe data manipulation: queries, updates, in sertions, and deletions, assuming a schema design has been provided. Schema design issues are deferred to Part 2.

- **Part 2: Database Design** (Chapters 7 through 9). Chapter 7 provides an overview of the database-design process, with major emphasis on database design using 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 that we encounter in capturing the semantics of realistic applica tions within the constraints of a data model. UML class-diagram notation is also covered in this chapter.

  Chapter 8 introduces the theory of relational database design. The the ory 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 intu itive 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. In structors may choose to use only this initial coverage in Sections 8.1 through 8.3 without loss of continuity. Instructors covering the entire chapter will ben efit from students having a good understanding of normalization concepts to motivate some of the challenging concepts of functional-dependency theory.

Chapter 9 covers application design and development. This chapter empha sizes the construction of database applications with Web-based interfaces. In addition, the chapter covers application security.

- **Part 3: Data Storage and Querying** (Chapters 10 through 13). Chapter 10 deals with storage devices, files, and data-storage structures. A variety of data-access techniques are presented in Chapter 11, including B^+-tree indices and hashing. Chapters 12 and 13 address query-evaluation algorithms and query optimization. These chapters provide an understanding of the internals of the storage and retrieval components of a database.

- **Part 4: Transaction Management** (Chapters 14 through 16). Chapter 14 fo cuses 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 locking and snapshot isolation.

  Chapter 15 focuses on concurrency control and presents several techniques for ensuring serializability, including locking, timestamping, and optimistic (validation) techniques. The chapter also covers deadlock issues. Alterna tives to serializability are covered, most notably the widely-used snapshot isolation, which is discussed in detail.

  Chapter 16 covers the primary techniques for ensuring correct transac tion execution despite system crashes and storage failures. These techniques include logs, checkpoints, and database dumps. The widely-used ARIES al gorithm is presented.

- **Part 5: System Architecture** (Chapters 17 through 19). Chapter 17 covers computer-system architecture, and describes the influence of the underly ing computer system on the database system. We discuss centralized sys tems, client–server systems, and parallel and distributed architectures in this chapter.

  Chapter 18, on parallel databases, explores a variety of parallelization techniques, including I/O parallelism, interquery and intraquery parallelism, and interoperation and intraoperation parallelism. The chapter also describes parallel-system design.

  Chapter 19 covers distributed database systems, revisiting the issues of database design, transaction management, and query evaluation and op timization, in the context of distributed databases. The chapter also cov ers issues of system availability during failures, heterogeneous distributed databases, cloud-based databases, and distributed directory systems.

- **Part 6: Data Warehousing, Data Mining, and Information Retrieval** (Chap ters 20 and 21). Chapter 20 introduces the concepts of data warehousing and data mining. Chapter 21 describes information-retrieval techniques for querying textual data, including hyperlink-based techniques used in Web search engines.

Part 6 uses the modeling and language concepts from Parts 1 and 2, but does not depend on Parts 3, 4, or 5. It can therefore be incorporated easily into a course that focuses on SQL and on database design.

- **Part 7: Specialty Databases** (Chapters 22 and 23). Chapter 22 covers object based databases. The chapter describes the object-relational data model, which extends the relational data model to support complex data types, type inheritance, and object identity. The chapter also describes database access from object-oriented programming languages.

  Chapter 23 covers the XML standard for data representation, which is seeing increasing use in the exchange and storage of complex data. The chapter also describes query languages for XML.

- **Part 8: Advanced Topics** (Chapters 24 through 26). Chapter 24 covers ad vanced issues in application development, including performance tuning, performance benchmarks, database-application testing, and standardization.

  Chapter 25 covers spatial and geographic data, temporal data, multimedia data, and issues in the management of mobile and personal databases. Finally, Chapter 26 deals with advanced transaction processing. Top ics covered in the chapter include transaction-processing monitors, transac tional workflows, electronic commerce, high-performance transaction sys tems, real-time transaction systems, and long-duration transactions.

- **Part 9: Case Studies** (Chapters 27 through 30). In this part, we present case studies of four of the leading database systems, PostgreSQL, Oracle, IBM DB2, and Microsoft SQL Server. These chapters outline unique features of each of these systems, and describe their internal structure. They provide a wealth of interesting information about the respective products, and help you see how the various implementation techniques described in earlier parts are used in real systems. They also cover several interesting practical aspects in the design of real systems.

- **Appendices**. We provide five appendices that cover material that is of histor ical nature or is advanced; these appendices are available only online on the Web site of the book (http://www.db-book.com). An exception is Appendix A, which presents details of our university schema including the full schema, DDL, and all the tables. This appendix appears in the actual text.

  Appendix B describes other relational query languages, including QBE Microsoft Access, and Datalog.

  Appendix C describes advanced relational database design, including the theory of multivalued dependencies, join dependencies, and the project-join and domain-key normal forms. This appendix is for the benefit of individuals who wish to study the theory of relational database design in more detail, and instructors who wish to do so in their courses. This appendix, too, is available only online, on the Web site of the book.

Although most new database applications use either the relational model or the object-relational model, the network and hierarchical data models are still in use in some legacy applications. For the benefit of readers who wish to learn about these data models, we provide appendices describing the network and hierarchical data models, in Appendices D and E respectively.

The Sixth Edition

The production of this sixth 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 have replaced the earlier running example of bank enterprise with a uni versity example. This example has an immediate intuitive connection to students that assists not only in remembering the example, but, more importantly, in gain ing deeper insight into the various design decisions that need to be made.

We have reorganized the book so as to collect all of our SQL coverage together and place it early in the book. Chapters 3, 4, and 5 present complete SQL coverage. Chapter 3 presents the basics of the language, with more advanced features in Chapter 4. In Chapter 5, we present JDBC along with other means of accessing SQL from a general-purpose programming language. We present triggers and re cursion, and then conclude with coverage of online analytic processing (OLAP). Introductory courses may choose to cover only certain sections of Chapter 5 or defer sections until after the coverage of database design without loss of continu ity.

Beyond these two major changes, we revised the material in each chapter, bringing the older material up-to-date, adding discussions on recent develop ments in database technology, and improving descriptions of topics that students found difficult to understand. We have also added new exercises and updated references. The list of specific changes includes the following:

- **Earlier coverage of SQL.** Many instructors use SQL as a key component of term projects (see our Web site, www.db-book.com, for sample projects). In order to give students ample time for the projects, particularly for universities and colleges on the quarter system, it is essential to teach SQL as early as possible. With this in mind, we have undertaken several changes in organization:
    - A new chapter on the relational model (Chapter 2) precedes SQL, laying the conceptual foundation, without getting lost in details of relational algebra.
    - Chapters 3, 4, and 5 provide detailed coverage of SQL. These chapters also discuss variants supported by different database systems, to

minimize problems that students face when they execute queries on actual database systems. These chapters cover all aspects of SQL, including queries, data definition, constraint specification, OLAP, and the use of SQL from within a variety of languages, including Java/JDBC.

◦ Formal languages (Chapter 6) have been postponed to after SQL, and can be omitted without affecting the sequencing of other chapters. Only our discussion of query optimization in Chapter 13 depends on the relational algebra coverage of Chapter 6.

- **New database schema.** We adopted a new schema, which is based on uni versity data, as a running example throughout the book. This schema is more intuitive and motivating for students than the earlier bank schema, and illustrates more complex design trade-offs in the database-design chapters.

- **More support for a hands-on student experience.** To facilitate following our running example, we list 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 http://www.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.

- **Revised coverage of E-R model.** The E-R diagram notation in Chapter 7 has been modified to make it more compatible with UML. The chapter also makes good use of the new university database schema to illustrate more complex design trade-offs.

- **Revised coverage of relational design.** Chapter 8 now has a more readable style, providing an intuitive understanding of functional dependencies and normalization, before covering functional dependency theory; the theory is motivated much better as a result.

- **Expanded material on application development and security.** Chapter 9 has new material on application development, mirroring rapid changes in the field. In particular, coverage of security has been expanded, considering its criticality in today's interconnected world, with an emphasis on practical issues over abstract concepts.

- **Revised and updated coverage of data storage, indexing and query op timization**. Chapter 10 has been updated with new technology, including expanded coverage of flash memory.

  Coverage of B^+-trees in Chapter 11 has been revised to reflect practical implementations, including coverage of bulk loading, and the presentation has been improved. The B^+-tree examples in Chapter 11 have now been revised with $n = 4$, to avoid the special case of empty nodes that arises with the (unrealistic) value of $n = 3$.

Chapter 13 has new material on advanced query-optimization techniques.

- **Revised coverage of transaction management.** Chapter 14 provides full coverage of the basics for an introductory course, with advanced details following in Chapters 15 and 16. Chapter 14 has been expanded to cover the practical issues in transaction management faced by database users and database application developers. The chapter also includes an expanded overview of topics covered in Chapters 15 and 16, ensuring that even if Chapters 15 and 16 are omitted, students have a basic knowledge of the concepts of concurrency control and recovery.

Chapters 14 and 15 now include detailed coverage of snapshot isolation, which is widely supported and used today, including coverage of potential hazards when using it.

Chapter 16 now has a simplified description of basic log-based recovery leading up to coverage of the ARIES algorithm.

- **Revised and expanded coverage of distributed databases.** We now cover cloud data storage, which is gaining significant interest for business applications. Cloud storage offers enterprises opportunities for improved cost management and increased storage scalability, particularly for Web-based applications. We examine those advantages along with the potential drawbacks and risks.

  Multidatabases, which were earlier in the advanced transaction processing chapter, are now covered earlier as part of the distributed database chapter.

- **Postponed coverage of object databases and XML.** Although object-oriented languages and XML are widely used outside of databases, their use in data bases is still limited, making them appropriate for more advanced courses, or as supplementary material for an introductory course. These topics have therefore been moved to later in the book, in Chapters 22 and 23.

- **QBE, Microsoft Access, and Datalog in an online appendix.** These topics, which were earlier part of a chapter on "other relational languages," are now covered in online Appendix C.

All topics not listed above are updated from the fifth edition, though their overall organization is relatively unchanged.

Review Material and Exercises

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

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

The book contains both basic and advanced material, which might not be cov ered in a single semester. We have marked several sections as advanced, using the symbol "**". These sections may be omitted if so desired, without a loss of continuity. Exercises that are difficult (and can be omitted) are also marked using the symbol "**".

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:

- Chapter 5 (Advanced SQL) 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 (Formal Relational Query Languages) can be covered immediately after Chapter 2, ahead of SQL. Alternatively, this chapter may be omitted from an introductory course.

  We recommend covering Section 6.1 (relational algebra) if the course also covers query processing. However, Sections 6.2 and 6.3 can be omitted if students will not be using relational calculus as part of the course.

- Chapter 7 (E-R Model) can be covered ahead of Chapters 3, 4 and 5 if you so desire, since Chapter 7 does not have any dependency on SQL.

- Chapter 13 (Query Optimization) can be omitted from an introductory course without affecting coverage of any other chapter.

- Both our coverage of transaction processing (Chapters 14 through 16) and our coverage of system architecture (Chapters 17 through 19) consist of an overview chapter (Chapters 14 and 17, respectively), followed by chapters with details. You might choose to use Chapters 14 and 17, while omitting Chapters 15, 16, 18 and 19, if you defer these latter chapters to an advanced course.

- Chapters 20 and 21, covering data warehousing, data mining, and informa tion retrieval, can be used as self-study material or omitted from an introduc tory course.

- Chapters 22 (Object-Based Databases), and 23 (XML) can be omitted from an introductory course.

- Chapters 24 through 26, covering advanced application development, spatial, temporal and mobile data, and advanced transaction processing, are suitable for an advanced course or for self-study by students.
- The case-study Chapters 27 through 30 are suitable for self-study by students. Alternatively, they can be used as an illustration of concepts when the earlier chapters are presented in class.

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: http://www.db-book.com. The Web site contains:

- Slides covering all the chapters of the book.
- Answers to the practice exercises.
- The five appendices.
- An up-to-date errata list.
- 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.

The following additional material is available only to faculty:

- An instructor 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 manual and the question bank, please send electronic mail to customer.service@mcgraw-hill.com. In the United States, you may call 800-338-3987. The McGraw-Hill Web site for this book is http://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 corre spondence 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 sixth edition, as well as with the previous five editions from which it is derived.

Sixth Edition

- Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, and Bianca Schroeder (Carnegie Mellon University) for writing Chapter 27 de scribing the PostgreSQL database system.
- Hakan Jakobsson (Oracle), for writing Chapter 28 on the Oracle database system.
- Sriram Padmanabhan (IBM), for writing Chapter 29 describing the IBM DB2 database system.
- Sameet Agarwal, Jose A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk My- ´ ers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas, and Michael Zwilling (all of Microsoft) for writing Chapter 30 de scribing the Microsoft SQL Server database system, and in particular Jose´ Blakeley for coordinating and editing the chapter; Cesar Galindo-Legaria, ´ Goetz Graefe, Kalen Delaney, and Thomas Casey (all of Microsoft) for their contributions to the previous edition of the Microsoft SQL Server chapter.
- 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 - Dallas; Sanjay Madria, University of Missouri - Rolla; 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 this sixth edition.
- Judi Paige for her help in generating figures and presentation slides.
- Mark Wogahn for making sure that the software to produce the book, includ ing LaTeX macros and fonts, worked properly.
- N. L. Sarda for feedback that helped us improve several chapters, in particular Chapter 11; 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.

Previous Editions

- 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 Dupre copyedited the third edition and Sara Strandtman edited the text ´ of the third edition.
- Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K. V. Ragha van, 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 pointed out errors in the fifth edition: Alex Coman, Ravindra Guravannavar, Arvind Hulgeri, Rohit Kulshreshtha, Sang-Won Lee, Joe H. C. Lu, Alex N. Napitupulu, H. K. Park, Jian Pei, Fernando Saenz Perez, Donnie Pinkston, Yma Pinto, Rajarshi Rakshit, Sandeep Satpal, Amon Seagull, Barry Soroka, Praveen Ranjan Srivastava, Hans Svensson, Moritz Wiese, and Eyob Delele Yirdaw.
- The following people offered suggestions and comments for the fifth and ear lier editions of the book. R. B. Abhyankar, Hani Abu-Salem, Jamel R. Alsab bagh, 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 Falout sos, 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 Riccardi, 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.

Book Production

The publisher was Raghu Srinivasan. The developmental editor was Melinda D. Bilecki. The project manager was Melissa Leick. The marketing manager was

Curt Reynolds. The production supervisor was Laura Fuller. The book designer was Brenda Rolwes. The cover designer was Studio Montage, St. Louis, Missouri. The copyeditor was George Watson. The proofreader was Kevin Campbell. The freelance indexer was Tobiah Waldron. The Aptara team consisted of Raman Arora and Sudeshna Nandy

Personal Notes

Sudarshan would like to acknowledge his wife, Sita, for her love 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.

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. Man agement 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 scien tists 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

Databases are widely used. Here are some representative applications:

- *Enterprise Information*
  - *Sales*: For customer, product, and purchase information.
  - *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 pro duction of items in factories, inventories of items in warehouses and stores, and orders for items.

  - *Online retailers*: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.
- *Banking and Finance*
  - *Banking*: For customer information, accounts, loans, and banking transac tions.
  - *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 made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

As the list illustrates, databases form an essential part of every enterprise today, storing not only types of information that are common to most enterprises, but also information that is specific to the category of the enterprise.

Over the course of the last four decades of the twentieth century, use of databases grew in all enterprises. In the early days, very few people interacted di rectly with database systems, although without realizing it, they interacted with databases indirectly— through printed reports such as credit card statements, or through agents such as bank tellers and airline reservation agents. Then auto mated teller machines came along and let users interact directly with databases. Phone interfaces to computers (interactive voice-response systems) also allowed users to deal directly with databases—a caller could dial a number, and press phone keys to enter information or to select alternative options, to find flight arrival/departure times, for example, or to register for courses in a university.

The Internet revolution of the late 1990s sharply increased direct user access to databases. Organizations converted many of their phone interfaces to databases into Web interfaces, and made a variety of services and information available online. For instance, when you 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, informa-

tion about you may be retrieved from a database to select which advertisements you should see. 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.

The importance of database systems can be judged in another way— today, database system vendors like Oracle are among the largest software companies in the world, and database systems form an important part of the product line of Microsoft and IBM.

1.2 Purpose of Database Systems

Database systems arose in response to early methods of computerized manage ment of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps infor mation 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

System programmers wrote 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 (say, computer science). As a result, the university creates a new department and creates new per manent 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, etc. 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 operat ing system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the ap propriate files. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems.

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

- **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 has now 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 cer tain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each ac count. 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 pro grams. 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 consistent state that existed prior to the failure. Consider a program to transfer \$500 from the account balance of department A to the account balance of department B. If a system failure occurs during the execution of the program, it is possible that the \$500 was removed from the balance of department A but was not credited to the balance of department 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 sys tem 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 environ ment, interaction of concurrent updates is possible and may result in incon sistent data. Consider department *A*, with an account balance of \$10,000. If two department clerks debit the account balance (by say \$500 and \$100, re spectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget 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 account balance of department *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 (say) 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 suc cessfully 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 applica tion programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database sys tems. 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 Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers 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 abstraction, to simplify users' interactions with the system:

- **Physical level**. The lowest level of abstraction describes *how* the data are ac tually 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 struc tures 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 independence**. 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 1.1 shows the relationship among the three levels of abstraction. An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming

physical
level

view level

view 1 view 2

logical
level

. . . view *n*

Figure 1.1 The three levels of data abstraction.

languages support the notion of a structured type. For example, we may describe a record 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. 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*

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive storage locations. 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.

[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 same effect.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. 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.

1.3.2 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**. Schemas are changed infrequently, if at all.

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 applications by using the logical 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 study languages for describing schemas after introducing the notion of data models in the next section.

1.3.3 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. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

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 repre sent both data and the relationships among those data. Each table has mul tiple 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 par ticular 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 major ity of current database systems are based on the relational model. Chapters 2 through 8 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, and Chapter 7 explores it in detail.

- **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model. Chap ter 22 examines the object-relational data model.

- **Semistructured Data Model**. The semistructured 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. The **Extensible Markup Language (XML)** is widely used to represent semistructured data. Chapter 23 covers it.

Historically, the **network data model** and the **hierarchical data model** pre ceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are used little now, except in old database code that is still in service in some places. They are outlined online in Appendices D and E for interested readers.

1.4 Database Languages

A database system provides a **data-definition language** to specify the database schema and a **data-manipulation language** to express database queries and up-

dates. In practice, the data-definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database lan guage, such as the widely used SQL language.

1.4.1 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:

- **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 Chap ters 3, 4, and 5. We also study some other query languages in Chapter 6.

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 Chapters 12 and 13) translates DML queries into sequences of actions at the physical level of the database system.

1.4.2 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 con straints**. 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 per taining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement 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 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 cer tain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. 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 con straint is violated, the normal procedure is to reject the action that caused the violation.
- **Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. 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.
- **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 mod ification 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 DDL, just like any other programming language, gets as input some instructions (statements) and 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 only be accessed and updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

1.5 Relational Databases

A relational database is based on the relational model and uses a collection of tables to represent both data and the relationships among those data. It also in cludes a DML and DDL. In Chapter 2 we present a gentle introduction to the fundamentals of the relational model. Most commercial relational database sys tems employ the SQL language, which we cover in great detail in Chapters 3, 4, and 5. In Chapter 6 we discuss other influential languages.

1.5.1 Tables

Each table has multiple columns and each column has a unique name. Figure 1.2 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.

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.

It is not hard to see how tables may be stored in files. For instance, a special character (such as a comma) may be 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. The relational model hides such low-level implementation details from database developers and users.

We also note that it is possible to create schemas in the relational model that have problems such as unnecessarily duplicated information. For example, sup pose we store the department *budget* as an attribute of the *instructor* record. Then, whenever the value of a particular budget (say that one for the Physics depart ment) changes, that change must to be reflected in the records of all instructors

ID name dept name salary

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

dept name	*building*	*budget*
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 1.2 A sample relational database.

associated with the Physics department. In Chapter 8, we shall study how to distinguish good schema designs from bad schema designs.

1.5.2 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. More specifically, 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 1.2, the result will consist 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 budget of greater 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 above query were run on the tables in Figure 1.2, the system would find that there are two departments with budget of greater than $95,000—Computer Science and Finance; there are five instructors in these departments. Thus, the result will consist 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).

1.5.3 Data-Definition Language

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc.

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 above 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. In addition, the DDL statement updates the data dictionary, which contains metadata (see Section 1.4.2). The schema of a table is an example of metadata.

1.5.4 Database Access from Application Programs

SQL is 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 com putations and actions must be written in a *host* language, such as C, C++, or Java, 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, etc. To access the database, DML statements need to be executed from the host language. There are two ways to do this:

- By providing 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 for use with the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.

- By extending the host language syntax to embed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the **DML precompiler**, converts the DML statements to normal procedure calls in the host language.

1.6 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 initially on the writing of database queries and the design of database schemas. Chapter 9 discusses the overall process of application design.

1.6.1 Design Process

A high-level data model provides the database designer with a conceptual frame work 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 de cisions 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 (Section 1.6.3); 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 (Section 1.6.4).

A fully developed conceptual schema 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 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 10.

1.6.2 Database Design for a University Organization

To illustrate the design process, let us examine how a database for a university could be designed. The initial specification of user requirements may be based on interviews with the database users, and on the designer's own analysis of the organization. The description that arises from this design phase serves as the basis for specifying the conceptual structure of the database. Here are the major characteristics of the university.

- The university is organized into departments. Each department is identified by a unique name (*dept name*), is located in a particular *building*, and has a *budget*.
- Each department has a list of courses it offers. Each course has associated with it a *course id*, *title*, *dept name*, and *credits*, and may also have have associated *prerequisites*.
- Instructors are identified by their unique *ID*. Each instructor has *name*, asso ciated department (*dept name*), and *salary*.
- Students are identified by their unique *ID*. Each student has a *name*, an associ ated major department (*dept name*), and *tot cred* (total credit hours the student earned thus far).

- The university maintains a list of classrooms, specifying the name of the *building*, *room number*, and room *capacity*.

- The university maintains a list of all classes (sections) taught. Each section is identified by a *course id*, *sec id*, *year*, and *semester*, and has associated with it a *semester*, *year*, *building*, *room number*, and *time slot id* (the time slot when the class meets).
- The department has a list of teaching assignments specifying, for each in structor, the sections the instructor is teaching.
- The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).

A real university database would be much more complex than the preceding design. However we use this simplified model to help you understand conceptual ideas without getting lost in details of a complex design.

1.6.3 The 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. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes *dept name*, *building*, and *budget* may describe one particular department in a university, and they form attributes of the *department* entity set. Similarly, attributes *ID*, *name*, and *salary* may describe an *instructor* entity.[2]

The extra attribute *ID* is used to identify an instructor uniquely (since it may be possible to have two instructors with the same name and the same salary). A unique instructor identifier must be assigned to each instructor. In the United States, many organizations use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a unique identifier.

A **relationship** is an association among several entities. For example, a *member* relationship associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

The overall logical structure (schema) of a database can be expressed graph ically by an *entity-relationship (E-R) diagram*. There are several ways in which to draw these diagrams. One of the most popular is to use the **Unified Modeling Language (UML)**. In the notation we use, which is based on UML, an E-R diagram is represented as follows:

[2]The astute reader will notice that we dropped the attribute *dept name* from the set of attributes describing the *instructor* entity set; this is not an error. In Chapter 7 we shall provide a detailed explanation of why this is the case.

ID
name
salary

instructor

dept_name *building*
budget

Figure 1.3 A sample E-R diagram.

~~*member*~~

department

- Entity sets are represented by a rectangular box with the entity set name in the header and the attributes listed below it.
- Relationship sets are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

As an illustration, consider part of a university database consisting of instruc tors and the departments with which they are associated. Figure 1.3 shows the corresponding E-R diagram. The E-R diagram indicates that there are two entity sets, *instructor* and *department*, with attributes as outlined earlier. The diagram also shows a relationship *member* between *instructor* and *department*.

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set. For example, if each instructor must be associated with only a single department, the E-R model can express that constraint.

The entity-relationship model is widely used in database design, and Chapter 7 explores it in detail.

1.6.4 Normalization

Another 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 to design schemas that are in an appropriate *normal form*. 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 8.4.

To understand the need for normalization, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information

- Inability to represent certain information

ID name salary dept name building budget

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 1.4 The *faculty* table.

We shall discuss these problems with the help of a modified database design for our university example.

Suppose that instead of having the two separate tables *instructor* and *depart ment*, we have a single table, *faculty*, that combines the information from the two tables (as shown in Figure 1.4). Notice that there are two rows in *faculty* that contain repeated information about the History department, specifically, that department's building and budget. The repetition of information in our alterna tive design is undesirable. Repeating information wastes space. Furthermore, it complicates updating the database. Suppose that we wish to change the budget amount of the History department from $50,000 to $46,800. This change must be reflected in the two rows; contrast this with the original design, where this requires an update to only a single row. Thus, updates are more costly under the alternative design than under the original design. When we perform the update in the alternative database, we must ensure that *every* tuple pertaining to the His tory department is updated, or else our database will show two different budget values for the History department.

Now, let us shift our attention to the issue of "inability to represent certain information." 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 rows in the faculty 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.

One solution to this problem is to introduce **null** values. The *null* value indicates that the value does not exist (or is not known). 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). As we shall see later, null values are difficult to handle, and it is preferable not to resort to them. If we are not willing to deal with null values, then we can create a particular item of department information only when the department has at least one instructor associated with the department. Furthermore, we would have to delete this information when the last instructor in the department departs. Clearly, this situation is undesirable, since, under our original database design, the department information would be available regardless of whether or not there is an instructor associated with the department, and without resorting to null values.

An extensive theory of normalization has been developed that helps formally define what database designs are undesirable, and how to obtain desirable de signs. Chapter 8 covers relational-database design, including normalization.

1.7 Data Storage and Querying

A database system is partitioned into modules that deal with each of the re sponsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is approxi mately 1000 megabytes (actually 1024) (1 billion bytes), and a terabyte is 1 million megabytes (1 trillion bytes). Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved be tween 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.

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.

1.7.1 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 respon sible 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 man ager 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 consis tent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- **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 phys ical 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*. Hashing is an alternative to indexing that is faster in some but not all cases.

We discuss storage media, file structures, and buffer management in Chapter 10. Methods of accessing data efficiently via indexing or hashing are discussed in Chapter 11.

1.7.2 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 12, while the methods by which the query optimizer chooses from among the possible evaluation strategies are discussed in Chapter 13.

1.8 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 department account (say *A*) is debited and another department account (say *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 preserve 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 ter minates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since 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 define properly the various transac tions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account of department *A* to the account of department *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 com plete 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, 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 con sistency of data may no longer be preserved, even though each individual transac tion is correct. It is the responsibility of the **concurrency-control manager** to con trol 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 14. The management of concurrent transactions is covered in Chapter 15. Chapter 16 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 applica tions, 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. These broader applications of the transaction concept are discussed in Chapter 26.

1.9 Database Architecture

We are now in a position to provide a single picture (Figure 1.5) of the various components of a database system and the connections among them.

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 client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit par allel computer architectures. Distributed databases span multiple geographically separated machines.

In Chapter 17 we cover the general structure of modern computer systems. Chapter 18 describes how various actions of a database, in particular query pro cessing, can be implemented to exploit parallel processing. Chapter 19 presents a number of issues that arise in 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. Distributed query processing and directory systems are also described in this chapter.

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between **client** machines, on which remote database users work, and **server** machines, on which the database system runs.

Database applications are usually partitioned into two or three parts, as in Figure 1.6. In a **two-tier architecture**, the application resides at the client machine, where it invokes database system functionality at the server machine through

(tellers, agents,
web users)
programmer
s
(analysts)
database
administrators
application
sophisticate
d users
naive users

use write use use
application programs

application
interfaces

compiler and linker
query
DML queries DDL
interpreter

toolsadministration

tools
DML compiler
and organizer

application
program
object code

manager

query evaluation
engine

storage manager disk storage

buffer manager file manager
authorization and integrity
manager
query processor

transaction

indices
data
dictionary

data statistical data **Figure 1.5** System structure.

network

user

application server

server

application

network

database system

user

client

database system

(a) Two-tier architecture

(b) Three-tier architecture

application client

Figure 1.6 Two-tier and three-tier architectures.

query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in a **three-tier architecture**, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an **application server**, usually through a forms interface. 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 are more appropriate for large applications, and for applications that run on the World Wide Web.

1.10 Data Mining and Information Retrieval

The term **data mining** refers loosely to the process of semiautomatically 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 machine learning and statistics in that it deals with large volumes of data, stored primarily on disk. That is, data mining deals with "knowledge discovery in databases."

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, but rather have

degrees of "support" and "confidence." Other types of knowledge are represented by equations relating different variables to each other, or by other mechanisms for predicting outcomes when the values of some variables are known.

There are a variety of possible types of patterns that may be useful, and different techniques are used to find different types of patterns. In Chapter 20 we 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 preprocess ing data to a form acceptable to the algorithms, and postprocessing 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.

Businesses have begun to exploit the burgeoning data online to make better decisions about their activities, such as what items to stock and how best to target customers to increase sales. Many of their queries are rather complicated, however, and certain types of information cannot be extracted even by using SQL.

Several techniques and tools are available to help with decision support. Several tools for data analysis allow analysts to view data in different ways. Other analysis tools precompute summaries of very large amounts of data, in order to give fast responses to queries. The SQL standard contains additional constructs to support data analysis.

Large companies have diverse sources of data that they need to use for making business decisions. To execute queries efficiently on such diverse data, companies have built *data warehouses*. Data warehouses gather data from multiple sources under a unified schema, at a single site. Thus, they provide the user a single uniform interface to data.

Textual data, too, has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as *information retrieval*. Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage. However, the emphasis in the field of information systems is different from that in database systems, concentrating on issues such as querying based on keywords; the relevance of documents to the query; and the analysis, classification, and indexing of documents. In Chapters 20 and 21, we cover decision support, including online analytical processing, data mining, data warehousing, and information retrieval.

1.11 Specialty Databases

Several application areas for database systems are limited by the restrictions of the relational data model. As a result, researchers have developed several data models to deal with these application domains, including object-based data models and semistructured data models.

1.11.1 Object-Based Data Models

Object-oriented programming has become the dominant software-development methodology. This led to the development of an **object-oriented data model** that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. Inheritance, object identity, and encapsulation (information hiding), with methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modeling. The object-oriented data model also supports a rich type system, including structured and collection types. In the 1980s, several database systems based on the object-oriented data model were developed.

The major database vendors presently support the **object-relational data model**, a data model that combines features of the object-oriented data model and relational data model. It extends the traditional relational model with a variety of features such as structured and collection types, as well as object orientation. Chapter 22 examines the object-relational data model.

1.11.2 Semistructured Data Models

Semistructured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes.

The XML language was initially designed as a way of adding markup infor mation to text documents, but has become important because of its applications in data exchange. XML provides a way to represent data that have nested structure, and furthermore allows a great deal of flexibility in structuring of data, which is important for certain kinds of nontraditional data. Chapter 23 describes the XML language, different ways of expressing queries on data represented in XML, and transforming XML data from one form to another.

1.12 Database Users and Administrators

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

1.12.1 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.

- **Na¨ıve users** are unsophisticated users who interact with the system by in voking one of the application programs that have been written previously. For example, a clerk in the university who needs to add a new instructor to

department *A* invokes a program called *new hire*. This program asks the clerk for the name of the new instructor, her new *ID*, the name of the department (that is, *A*), and the salary.

The typical user interface for na¨ıve users is a forms interface, where the user can fill in appropriate fields of the form. Na¨ıve users may also simply read *reports* generated from the database.

As another 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 appli cation first verifies the identity of the user, and 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. **Rapid application development (RAD)** tools are tools that en able an application programmer to construct forms and reports with minimal programming effort.

- **Sophisticated users** interact with the system without writing programs. In stead, 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.

- **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems. Chapter 22 covers several of these applications.

1.12.2 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 execut ing a set of data definition statements in the DDL.
- **Storage structure and access-method definition**.
- **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 someone attempts to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
    - Periodically backing up the database, either onto tapes or 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.

1.13 History of Database Systems

Information processing drives the growth of computers, as it has from the earli est 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 process data in a particular order, by reading and merging data from tapes and card decks.

- **Late 1960s and 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 disks, network and hierarchical databases could be created that 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 Codd [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.

- **1980s**: Although academically interesting, the relational model was not used in practice initially, because of its perceived performance disadvantages; rela tional databases could not match the performance of existing network and hi erarchical databases. That changed with System R, a groundbreaking project at IBM Research that developed techniques for the construction of an efficient relational database system. Excellent overviews of System R are provided by Astrahan et al. [1976] and Chamberlin et al. [1981]. The fully functional Sys tem 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. Ini tial commercial relational database systems, such as IBM

DB2, Oracle, Ingres, and DEC Rdb, played a major role in advancing techniques for efficient pro cessing 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 such databases were forced to deal with many low-level implementation de tails, and 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, 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.

- **Early 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. Decision support and querying re-emerged as a major application area for databases. Tools for analyzing large amounts of data saw large growths in usage.

  Many database vendors introduced parallel database products in this period. Database vendors also began to add object-relational support to their databases.

- **1990s**: The major event of the 1990s was the explosive growth of the World WideWeb. 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 first half of the 2000s saw the emerging of XML and the associated query language XQuery as a new database technology. Although XML is widely used for data exchange, as well as for storing certain complex data types, relational databases still form the core of a vast majority of large-scale database applications. In this time period we have also witnessed the growth in "autonomic-computing/auto-admin" techniques for minimizing system administration effort. This period also saw a significant growth in use of open-source database systems, particularly PostgreSQL and MySQL.

  The latter part of the decade has seen growth in specialized databases for data analysis, in particular column-stores, which in effect store each column of a table as a separate array, and highly parallel database systems designed for analysis of very large data sets. Several novel distributed data-storage systems have been built to handle the data management requirements of very large Web sites such as Amazon, Facebook, Google, Microsoft and Yahoo!, and some of these are now offered as Web services that can be used by application developers. There has also been substantial work on management and analysis of streaming data, such as stock-market ticker data or computer network monitoring data. Data-mining techniques are now widely deployed; example applications include Web-based product-recommendation systems and automatic placement of relevant advertisements on Web pages.

1.14 Summary

- A **database-management system** (DBMS) consists of a collection of interre lated data and a collection of programs to access that data. The data describe one particular enterprise.
- The primary goal of a DBMS is to provide an environment that is both conve nient and efficient for people to use in retrieving and storing information.
- Database systems are ubiquitous today, and most people interact, either di rectly or indirectly, with databases many times every day.
- Database systems are designed to store large bodies of information. The man agement of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of infor mation. 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 semistructured 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 as well as 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 data model for database design. It provides a convenient graphical representation to view data, rela tionships, 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.
  - 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 conflicting.

- 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 client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. 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 back end. 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.

- Knowledge-discovery techniques attempt to discover automatically statisti cal rules and patterns from data. The field of**datamining** combines knowledge discovery techniques invented by artificial intelligence researchers and sta tistical analysts, with efficient implementation techniques that enable them to be used on extremely large databases.

- 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.

Review Terms

- Database-management system (DBMS)
- Database-system applications
- File-processing systems
- Data inconsistency
- Consistency constraints
- Data abstraction
- Instance
- Schema

- Physical schema
- Logical schema

- Physical data independence
- Data models
  - Entity-relationship model
  - Relational data model
  - Object-based data model
  - Semistructured data model
- Database languages
  - Data-definition language
  - Data-manipulation language
  - Query language
- Metadata
- Application program
- Normalization
- Data dictionary
- Storage manager
- Query processor
- Transactions
  - Atomicity
  - Failure recovery
  - Concurrency control
- Two- and three-tier database archi tectures
- Data mining
- 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.

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

1.4 List at least 3 different types of information that a university would main tain, beyond those listed in Section 1.6.2.

1.5 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.

1.6 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

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

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

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

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

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

1.12 Explain what problems are caused by the design of the table in Figure 1.4.

1.13 What are five main functions of a database administrator?

1.14 Explain the difference between two-tier and three-tier architectures. Which is better suited for Web applications? Why?

1.15 Describe at least 3 tables that might be used to store information in a social-networking system such as Facebook.

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), Sybase (www.sybase.com), and IBM Informix (www.ibm.com/software/data/informix). Some of these systems are available

free for personal or noncommercial 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) and PostgreSQL (www.postgresql.org). A more complete list of links to vendor Web sites and other information is available from the home page of this book, at www.db-book.com.

Bibliographical Notes

We list below general-purpose books, research paper collections, and Web sites on databases. Subsequent chapters provide references to material on each topic outlined in this chapter.

Codd [1970] is the landmark paper that introduced the relational model. Textbooks covering database systems include Abiteboul et al. [1995], O'Neil and O'Neil [2000], Ramakrishnan and Gehrke [2002], Date [2003], Kifer et al. [2005], Elmasri and Navathe [2006], and Garcia-Molina et al. [2008]. Textbook coverage of transaction processing is provided by Bernstein and Newcomer [1997] and

Gray and Reuter [1993]. A book containing a collection of research papers on database management is offered by Hellerstein and Stonebraker [2005]. A review of accomplishments in database management and an assessment of future research challenges appears in Silberschatz et al. [1990], Silberschatz et al. [1996], Bernstein et al. [1998], Abiteboul et al. [2003], and Agrawal et al. [2009]. 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.

This page intentionally left blank

PART 1

RELATIONAL DATABASES

A data model is a collection of conceptual tools for describing data, data relation ships, data semantics, and consistency constraints. In this part, we focus on the relational model.

The relational model, which is covered in Chapter 2, 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. The entity-relationship model, discussed later in Chapter 7 (in Part 2), is a higher-level data model which is widely used for database design.

To make data from a relational database available to users, we have to ad dress several issues. The most important issue is how users specify requests for retrieving and updating data; several query languages have been developed for this task. A second, but still important, issue is data integrity and protection; databases need to protect data from damage by user actions, whether uninten tional or intentional.

Chapters 3, 4 and 5 cover the SQL language, which is the most widely used query language today. Chapters 3 and 4 provide introductory and intermediate level descriptions of SQL. Chapter 4 also covers integrity constraints which are enforced by the database, and authorization mechanisms, which control what access and update actions can be carried out by a user. Chapter 5 covers more advanced topics, including access to SQL from programming languages, and the use of SQL for data analysis.

Chapter 6 covers three formal query languages, the relational algebra, the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic. These formal languages form the basis for SQL, and for two other user-friendly languages, QBE and Datalog, which are described in Appendix B (available online at db-book.com).

This page intentionally left blank

CHAPTER 2

Introduction to the Relational Model

The relational model is today 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.

In this chapter, we first study the fundamentals of the relational model. A substantial theory exists for relational databases. We study the part of this theory dealing with queries in Chapter 6. In Chapters 7 through 8, we shall examine aspects of database theory that help in the design of relational database schemas, while in Chapters 12 and 13 we discuss aspects of the theory dealing with efficient processing of queries.

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*.

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, we consider the table *instructor*, a row in the table can be thought of as representing

ID	*name*	*dept name*	*salary*
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.

the relationship 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 repre sented mathematically by an *n-tuple* of values, i.e., a tuple with n values, which corresponds to a row in a table.

course id	*title*	*dept name*	*credits*
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.

course id prereq id

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.

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, i.e., 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. They do not include all the data an actual university database would contain, in order to simplify our presentation. We shall discuss criteria for the appropriateness of relational structures in great detail in Chapters 7 and 8.

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

ID	*name*	*dept name*	*salary*
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.

the two figures are the same, since both contain the same set of tuples. For ease of exposition, we will mostly 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.

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 nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone number* would have an atomic domain.

In this chapter, as well as in Chapters 3 through 6, we assume that all attributes have atomic domains. In Chapter 22, we shall discuss extensions to the relational data model to permit nonatomic domains.

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 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.

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 no tion 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 corre sponding 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;

dept name	*building*	*budget*
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.

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

Let us continue with our university database example.

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 individ ual 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*)

course id	*sec id*	*semester*	*year*	*building*	*room number*	*time slot id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 2.6 The *section* relation.

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

As you can imagine, there are many more relations maintained in a real uni versity 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:

ID	*course id*	*sec id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010

45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 2.7 The *teaches* relation.

- *student* (*ID*, *name*, *dept name*, *tot cred*)
- *advisor* (*s id*, *i id*)
- *takes* (*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
- *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 distin guished. 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.

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 $t_1 = t_2$, then $t_1.K = t_2.K$.

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.

Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, 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 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 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.

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. A relation, say r_1, may include among its attributes the primary key of another relation, say r_2. This attribute 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 dependency, and r_2 is called the **referenced relation** of the foreign key. For example, the attribute *dept name* in *instructor* is a foreign key from *instructor*, referencing *depart ment*, since *dept name* is the primary key of *department*. 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.

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one instructor; however, it could possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; 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.

2.4 Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 2.8 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 dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

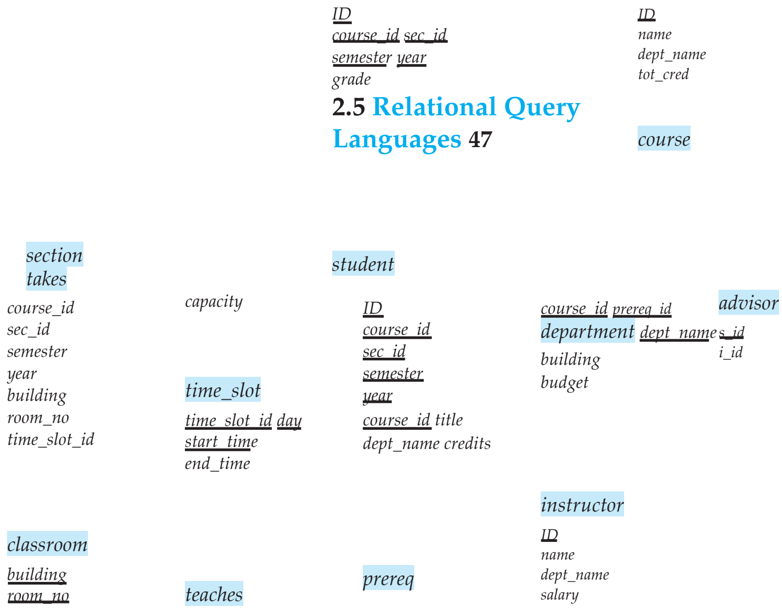

Figure 2.8 Schema diagram for the university database.

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram later, in Chapter 7. Entity relationship diagrams let us represent several kinds of constraints, including general referential integrity constraints.

Many database systems provide design tools with a graphical user interface for creating schema diagrams. We shall discuss diagrammatic representation of schemas at length in Chapter 7.

The enterprise that we use in the examples in later chapters is a university. Figure 2.9 gives the relational schema that we use in our examples, with primary key attributes underlined. As we shall see in Chapter 3, this corresponds to the approach to defining relations in the SQL data-definition language.

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 either procedural or nonprocedural. In a **procedural language**, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a **nonprocedural language**, the user describes the desired information without giving a specific procedure for obtaining that information.

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 2.9 Schema of the university database.

Query languages used in practice include elements of both the procedural and the nonprocedural approaches. We study the very widely used query language SQL in Chapters 3 through 5.

There are a number of "pure" query languages: The relational algebra is pro cedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural. 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. In Chapter 6, we examine in detail the rela tional algebra and the two versions of the relational calculus, the tuple relational calculus and domain relational calculus. 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. The relational calculus uses predicate logic to define the result desired without giving any specific algebraic procedure for obtaining that result.

2.6 Relational Operations

All procedural relational query languages provide a set of operations that can be applied to either a single relation or a pair of relations. These operations have the nice and desired property that their result is always a single relation. This property allows one to combine several of these operations in a modular way. Specifically, since the result of a relational query is itself a relation, relational operations can be applied to the results of queries as well as to the given set of relations.

The specific relational operations are expressed differently depending on the language, but fit the general framework we describe in this section. In Chapter 3, we show the specific way the operations are expressed in SQL.

The most frequent operation is the selection of specific tuples from a sin gle relation (say *instructor*) that satisfies some particular predicate (say *salary* > \$85,000). The result is a new relation that is a subset of the original relation (*in-*

ID	*name*	*dept name*	*salary*
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

Figure 2.10 Result of query selecting *instructor* tuples with salary greater than \$85000.

structor). For example, if we select tuples from the *instructor* relation of Figure 2.1, satisfying the predicate "*salary* is greater than \$85000", we get the result shown in Figure 2.10.

Another frequent operation is to select certain attributes (columns) from a relation. The result is a new relation having only those selected attributes. For example, suppose we want a list of instructor *ID*s and salaries without listing the *name* and *dept name* values from the *instructor* relation of Figure 2.1, then the result, shown in Figure 2.11, has the two attributes *ID* and *salary*. Each tuple in the result is derived from a tuple of the *instructor* relation but with only selected attributes shown.

The *join* operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple. There are a number of different ways to join relations (as we shall see in Chapter 3). Figure 2.12 shows an example of joining the tuples from the *instructor* and *department* tables with the new tuples showing the information about each instructor and the department in which she is working. This result was formed by combining each tuple in the *instructor* relation with the tuple in the *department* relation for the instructor's department.

In the form of join shown in Figure 2.12, which is called a *natural join*, a tuple from the *instructor* relation matches a tuple in the *department* relation if the values

ID	*salary*
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

Figure 2.11 Result of query selecting attributes *ID* and *salary* from the *instructor* relation.

ID	*name*	*salary*	*dept name*	*building*	*budget*
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Figure 2.12 Result of natural join of the *instructor* and *department* relations.

of their *dept name* attributes are the same. All such matching pairs of tuples are present in the join result. In general, the natural join operation on two relations matches tuples whose values are the same on all attribute names that are common to both relations.

The *Cartesian product* operation combines tuples from two relations, but unlike the join operation, its result contains *all* pairs of tuples from the two relations, regardless of whether their attribute values match.

Because relations are sets, we can perform normal set operations on relations. The *union* operation performs a set union of two "similarly structured" tables (say a table of all graduate students and a table of all undergraduate students). For example, one can obtain the set of all students in a department. Other set operations, such as *intersection* and *set difference* can be performed as well.

As we noted earlier, we can perform operations on the results of queries. For example, if we want to find the *ID* and *salary* for those instructors who have salary greater than \$85,000, we would perform the first two operations in our example above. First we select those tuples from the *instructor* relation where the *salary* value is greater than \$85,000 and then, from that result, select the two attributes *ID* and *salary*, resulting in the relation shown in Figure 2.13 consisting of the *ID*

ID	*salary*
12121	90000
22222	95000
33456	87000
83821	92000

Figure 2.13 Result of selecting attributes *ID* and *salary* of instructors with salary greater than \$85,000.

RELATIONAL ALGEBRA

The relational algebra defines a set of operations on relations, paralleling the usual algebraic operations such as addition, subtraction or multiplication, which operate on numbers. Just as algebraic operations on numbers take one or more numbers as input and return a number as output, the relational algebra op erations typically take one or two relations as input and return a relation as output.

Relational algebra is covered in detail in Chapter 6, but we outline a few of the operations below.

Symbol (Name) Example of Use

salary$_{>=85000}$(*instructor*)

(Selection) Return rows of the input relation that satisfy the predicate.

$_{ID,salary}$ (*instructor*)

(Projection) Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.

✶ *instructor* ✶ *department*

(Natural join) Output pairs of rows from the two input rela tions that have the same value on all attributes that have the same name.

× *instructor* × *department*

(Cartesian product) Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)

∪ $_{name}$ (*instructor*) ∪ $_{name}$ (*student*)

(Union) Output the union of tuples from the two input relations.

and *salary*. In this example, we could have performed the operations in either order, but that is not the case for all situations, as we shall see. Sometimes, the result of a query contains duplicate tuples. For example, if we select the *dept name* attribute from the *instructor* relation, there are several cases of duplication, including "Comp. Sci.", which shows up three times. Certain rela tional languages adhere strictly to the mathematical definition of a set and remove duplicates. Others, in consideration of the relatively large amount of processing required to remove duplicates from large result relations, retain duplicates. In these latter cases, the relations are not truly relations in the pure mathematical sense of the term.

Of course, data in a database must be changed over time. A relation can be updated by inserting new tuples, deleting existing tuples, or modifying tuples by

changing the values of certain attributes. Entire relations can be deleted and new ones created.

We shall discuss relational queries and updates using the SQL language in Chapters 3 through 5.

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 in cludes 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** is a set of attributes in a referencing relation, such that for each tuple in the referencing relation, the values of the foreign key attributes are guaranteed to occur as the primary key value of a tuple in the referenced relation.
- 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 add a number of useful syntactic features.

Review Terms

- Table
- Relation
- Tuple
- Attribute
- Domain
- Atomic domain

- Null value
- Database schema
- Database instance
- Relation schema
- Relation instance
- Keys
  - Superkey
  - Candidate key
  - Primary key
- Foreign key
  - Referencing relation
  - Referenced relation
- Referential integrity constraint
- Schema diagram
- Query language
  - Procedural language
  - Nonprocedural language
- Operations on relations
  - Selection of tuples
  - Selection of attributes
  - Natural join
  - Cartesian product
  - Set operations
- Relational algebra

Practice Exercises

2.1 Consider the relational database of Figure 2.14. 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, which 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 cross 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 $_{s\ id=I\ D}$(*student* × *advisor*).)

employee (*person name*, *street*, *city*)
works (*person name*, *company name*, *salary*)
company (*company name*, *city*)

Figure 2.14 Relational database for Exercises 2.1, 2.7, and 2.12.