FUNDAMENTALS OF

DATABASE SYSTEMS

7TH Edition

ELMASRI • NAVATHE

FUNDAMENTALS OF

Database Systems

SEVENTH EDITION

This page intentionally left blank

FUNDAMENTALS OF

Database Systems

SEVENTH EDITION

Ramez Elmasri
Department of Computer Science and Engineering
The University of Texas at Arlington

Shamkant B. Navathe
College of Computing
Georgia Institute of Technology

To Amalia
and
to Ramy, Riyad, Katrina, and Thomas

R. E.

To my wife Aruna for her love, support, and understanding
and
to Rohan, Maya, and Ayush for bringing so much joy into our lives

S.B.N.

This page intentionally left blank

Preface

This book introduces the fundamental concepts necessary for designing, using, and implementing database systems and database applications. Our presentation stresses the fundamentals of database modeling and design, the languages and models provided by the database management systems, and database system implementation techniques. The book is meant to be used as a textbook for a one- or two-semester course in database systems at the junior, senior, or graduate level, and as a reference book. Our goal is to provide an in-depth and up-to-date presentation of the most important aspects of database systems and applications, and related technologies. We assume that readers are familiar with elementary programming and data-structuring concepts and that they have had some exposure to the basics of computer organization.

New to This Edition

The following key features have been added in the seventh edition:

- A reorganization of the chapter ordering (this was based on a survey of the instructors who use the textbook); however, the book is still organized so that the individual instructor can choose to follow the new chapter ordering or *choose a different ordering of chapters* (for example, follow the chapter order from the sixth edition) when presenting the materials.

- There are two new chapters on recent advances in database systems and big data processing; one new chapter (Chapter 24) covers an introduction to the newer class of database systems known as **NOSQL databases**, and the other new chapter (Chapter 25) covers technologies for processing **big data**, including **MapReduce** and **Hadoop.**

- The chapter on query processing and optimization has been expanded and reorganized into two chapters; Chapter 18 focuses on strategies and algorithms for query processing whereas Chapter 19 focuses on query optimization techniques.

- A second UNIVERSITY database example has been added to the early chapters (Chapters 3 through 8) in addition to our COMPANY database example from the previous editions.

- Many of the individual chapters have been updated to varying degrees to include newer techniques and methods; rather than discuss these enhancements here,

we will describe them later in the preface when we discuss the organization of the seventh edition.

The following are key features of the book:

- A self-contained, flexible organization that can be tailored to individual needs; in particular, *the chapters can be used in different orders* depending on the instructor's preference.

- A companion website (http://www.pearsonhighered.com/cs-resources) includes data to be loaded into various types of relational databases for more realistic student laboratory exercises.

- A dependency chart (shown later in this preface) to show which chapters depend on other earlier chapters; this can guide the instructor who wants to tailor the *order of presentation of the chapters*.

- A collection of supplements, including a robust set of materials for instructors and students such as PowerPoint slides, figures from the text, and an instructor's guide with solutions.

Organization and Contents of the Seventh Edition

There are some organizational changes in the seventh edition as well as improvement to the individual chapters. The book is now divided into 12 parts as follows:

- Part 1 (Chapters 1 and 2) describes the basic introductory concepts necessary for a good understanding of database models, systems, and languages. Chapters 1 and 2 introduce databases, typical users, and DBMS concepts, terminology, and architecture, as well as a discussion of the progression of database technologies over time and a brief history of data models. These chapters have been updated to introduce some of the newer technologies such as NOSQL systems.

- Part 2 (Chapters 3 and 4) includes the presentation on entity-relationship modeling and database design; however, it is *important to note* that instructors can cover the relational model chapters (Chapters 5 through 8) *before Chapters 3 and 4* if that is their preferred order of presenting the course materials. In Chapter 3, the concepts of the Entity-Relationship (ER) model and ER diagrams are presented and used to illustrate conceptual database design. Chapter 4 shows how the basic ER model can be extended to incorporate additional modeling concepts such as subclasses, specialization, generalization, union types (categories) and inheritance, leading to the enhanced-ER (EER) data model and EER diagrams. The notation for the class diagrams of UML are also introduced in Chapters 7 and 8 as an alternative model and diagrammatic notation for ER/EER diagrams.

- Part 3 (Chapters 5 through 8) includes a detailed presentation on relational databases and SQL with some additional new material in the SQL chapters to cover a few SQL constructs that were not in the previous edition. Chapter 5

describes the basic relational model, its integrity constraints, and update operations. Chapter 6 describes some of the basic parts of the SQL standard for relational databases, including data definition, data modification operations, and simple SQL queries. Chapter 7 presents more complex SQL queries, as well as the SQL concepts of triggers, assertions, views, and schema modification. Chapter 8 describes the formal operations of the relational algebra and introduces the relational calculus. The material on SQL (Chapters 6 and 7) is presented before our presentation on relational algebra and calculus in Chapter 8 to allow instructors to start SQL projects early in a course if they wish (it is possible to cover Chapter 8 before Chapters 6 and 7 if the instructor desires this order). The final chapter in Part 2, Chapter 9, covers ER- and EER-to-relational mapping, which are algorithms that can be used for designing a relational database schema from a conceptual ER/EER schema design.

- Part 4 (Chapters 10 and 11) are the chapters on database programming techniques; these chapters can be assigned as reading materials and augmented with materials on the particular language used in the course for programming projects (much of this documentation is readily available on the Web). Chapter 10 covers traditional SQL programming topics, such as embedded SQL, dynamic SQL, ODBC, SQLJ, JDBC, and SQL/CLI. Chapter 11 introduces Web database programming, using the PHP scripting language in our examples, and includes new material that discusses Java technologies for Web database programming.

- Part 5 (Chapters 12 and 13) covers the updated material on object-relational and object-oriented databases (Chapter 12) and XML (Chapter 13); both of these chapters now include a presentation of how the SQL standard incorporates object concepts and XML concepts into more recent versions of the SQL standard. Chapter 12 first introduces the concepts for object databases, and then shows how they have been incorporated into the SQL standard in order to add object capabilities to relational database systems. It then covers the ODMG object model standard, and its object definition and query languages. Chapter 13 covers the XML (eXtensible Markup Language) model and languages, and discusses how XML is related to database systems. It presents XML concepts and languages, and compares the XML model to traditional database models. We also show how data can be converted between the XML and relational representations, and the SQL commands for extracting XML documents from relational tables.

- Part 6 (Chapters 14 and 15) are the normalization and relational design theory chapters (we moved all the formal aspects of normalization algorithms to Chapter 15). Chapter 14 defines functional dependencies, and the normal forms that are based on functional dependencies. Chapter 14 also develops a step-by-step intuitive normalization approach, and includes the definitions of multivalued dependencies and join dependencies. Chapter 15 covers normalization theory, and the formalisms, theories,

and algorithms developed for relational database design by normalization, including the relational decomposition algorithms and the relational synthesis algorithms.

- Part 7 (Chapters 16 and 17) contains the chapters on file organizations on disk (Chapter 16) and indexing of database files (Chapter 17). Chapter 16 describes primary methods of organizing files of records on disk, including ordered (sorted), unordered (heap), and hashed files; both static and dynamic hashing techniques for disk files are covered. Chapter 16 has been updated to include materials on buffer management strategies for DBMSs as well as an overview of new storage devices and standards for files and modern storage architectures. Chapter 17 describes indexing techniques for files, including B-tree and B$^+$-tree data structures and grid files, and has been updated with new examples and an enhanced discussion on indexing, including how to choose appropriate indexes and index creation during physical design.

- Part 8 (Chapters 18 and 19) includes the chapters on query processing algorithms (Chapter 18) and optimization techniques (Chapter 19); these two chapters have been updated and reorganized from the single chapter that covered both topics in the previous editions and include some of the newer techniques that are used in commercial DBMSs. Chapter 18 presents algorithms for searching for records on disk files, and for joining records from two files (tables), as well as for other relational operations. Chapter 18 contains new material, including a discussion of the semi-join and anti-join operations with examples of how they are used in query processing, as well as a discussion of techniques for selectivity estimation. Chapter 19 covers techniques for query optimization using cost estimation and heuristic rules; it includes new material on nested subquery optimization, use of histograms, physical optimization, and join ordering methods and optimization of typical queries in data warehouses.

- Part 9 (Chapters 20, 21, and 22) covers transaction processing concepts; concurrency control; and database recovery from failures. These chapters have been updated to include some of the newer techniques that are used in some commercial and open source DBMSs. Chapter 20 introduces the techniques needed for transaction processing systems, and defines the concepts of recoverability and serializability of schedules; it has a new section on buffer replacement policies for DBMSs and a new discussion on the concept of snapshot isolation. Chapter 21 gives an overview of the various types of concurrency control protocols, with a focus on two-phase locking. We also discuss timestamp ordering and optimistic concurrency control techniques, as well as multiple-granularity locking. Chapter 21 includes a new presentation of concurrency control methods that are based on the snapshot isolation concept. Finally, Chapter 23 focuses on database recovery protocols, and gives an overview of the concepts and techniques that are used in recovery.

■ Part 10 (Chapters 23, 24, and 25) includes the chapter on distributed databases (Chapter 23), plus the two new chapters on NOSQL storage systems for big data (Chapter 24) and big data technologies based on Hadoop and MapReduce (Chapter 25). Chapter 23 introduces distributed database concepts, including availability and scalability, replication and fragmentation of data, maintaining data consistency among replicas, and many other concepts and techniques. In Chapter 24, NOSQL systems are categorized into four general categories with an example system in each category used for our examples, and the data models, operations, as well as the replication/distribution/scalability strategies of each type of NOSQL system are discussed and compared. In Chapter 25, the MapReduce programming model for distributed processing of big data is introduced, and then we have presentations of the Hadoop system and HDFS (Hadoop Distributed File System), as well as the Pig and Hive high-level interfaces, and the YARN architecture.

■ Part 11 (Chapters 26 through 29) is entitled Advanced Database Models, Systems, and Applications and includes the following materials: Chapter 26 introduces several advanced data models including active databases/triggers (Section 26.1), temporal databases (Section 26.2), spatial databases (Section 26.3), multimedia databases (Section 26.4), and deductive databases (Section 26.5). Chapter 27 discusses information retrieval (IR) and Web search, and includes topics such as IR and keyword-based search, comparing DB with IR, retrieval models, search evaluation, and ranking algorithms. Chapter 28 is an introduction to data mining including overviews of various data mining methods such as associate rule mining, clustering, classification, and sequential pattern discovery. Chapter 29 is an overview of data warehousing including topics such as data warehousing models and operations, and the process of building a data warehouse.

■ Part 12 (Chapter 30) includes one chapter on database security, which includes a discussion of SQL commands for discretionary access control (GRANT, REVOKE), as well as mandatory security levels and models for including mandatory access control in relational databases, and a discussion of threats such as SQL injection attacks, as well as other techniques and methods related to data security and privacy.

Appendix A gives a number of alternative diagrammatic notations for displaying a conceptual ER or EER schema. These may be substituted for the notation we use, if the instructor prefers. Appendix B gives some important physical parameters of disks. Appendix C gives an overview of the QBE graphical query language, and Appendixes D and E (available on the book's Companion Website located at http://www.pearsonhighered.com/elmasri) cover legacy database systems, based on the hierarchical and network database models. They have been used for more than thirty years as a basis for many commercial database applications and transaction-processing systems.

Guidelines for Using This Book

There are many different ways to teach a database course. The chapters in Parts 1 through 7 can be used in an introductory course on database systems in the order that they are given or in the preferred order of individual instructors. Selected chapters and sections may be left out and the instructor can add other chapters from the rest of the book, depending on the emphasis of the course. At the end of the opening section of some of the book's chapters, we list sections that are candidates for being left out whenever a less-detailed discussion of the topic is desired. We suggest covering up to Chapter 15 in an introductory database course and including selected parts of other chapters, depending on the background of the students and the desired coverage. For an emphasis on system implementation techniques, chapters from Parts 7, 8, and 9 should replace some of the earlier chapters.

Chapters 3 and 4, which cover conceptual modeling using the ER and EER models, are important for a good conceptual understanding of databases. However, they may be partially covered, covered later in a course, or even left out if the emphasis is on DBMS implementation. Chapters 16 and 17 on file organizations and indexing may also be covered early, later, or even left out if the emphasis is on database models and languages. For students who have completed a course on file organization, parts of these chapters can be assigned as reading material or some exercises can be assigned as a review for these concepts.

If the emphasis of a course is on database design, then the instructor should cover Chapters 3 and 4 early on, followed by the presentation of relational databases. A total life-cycle database design and implementation project would cover conceptual design (Chapters 3 and 4), relational databases (Chapters 5, 6, and 7), data model mapping (Chapter 9), normalization (Chapter 14), and application programs implementation with SQL (Chapter 10). Chapter 11 also should be covered if the emphasis is on Web database programming and applications. Additional documentation on the specific programming languages and RDBMS used would be required. The book is written so that it is possible to cover topics in various sequences. The following chapter dependency chart shows the major dependencies among chapters. As the diagram illustrates, it is possible to start with several different topics following the first two introductory chapters. Although the chart may seem complex, it is important to note that if the chapters are covered in order, the dependencies are not lost. The chart can be consulted by instructors wishing to use an alternative order of presentation.

For a one-semester course based on this book, selected chapters can be assigned as reading material. The book also can be used for a two-semester course sequence. The first course, *Introduction to Database Design and Database Systems*, at the sophomore, junior, or senior level, can cover most of Chapters 1 through 15. The second course, *Database Models and Implementation Techniques*, at the senior or first-year graduate level, can cover most of Chapters 16 through 30. The two-semester sequence can also be designed in various other ways, depending on the preferences of the instructors.
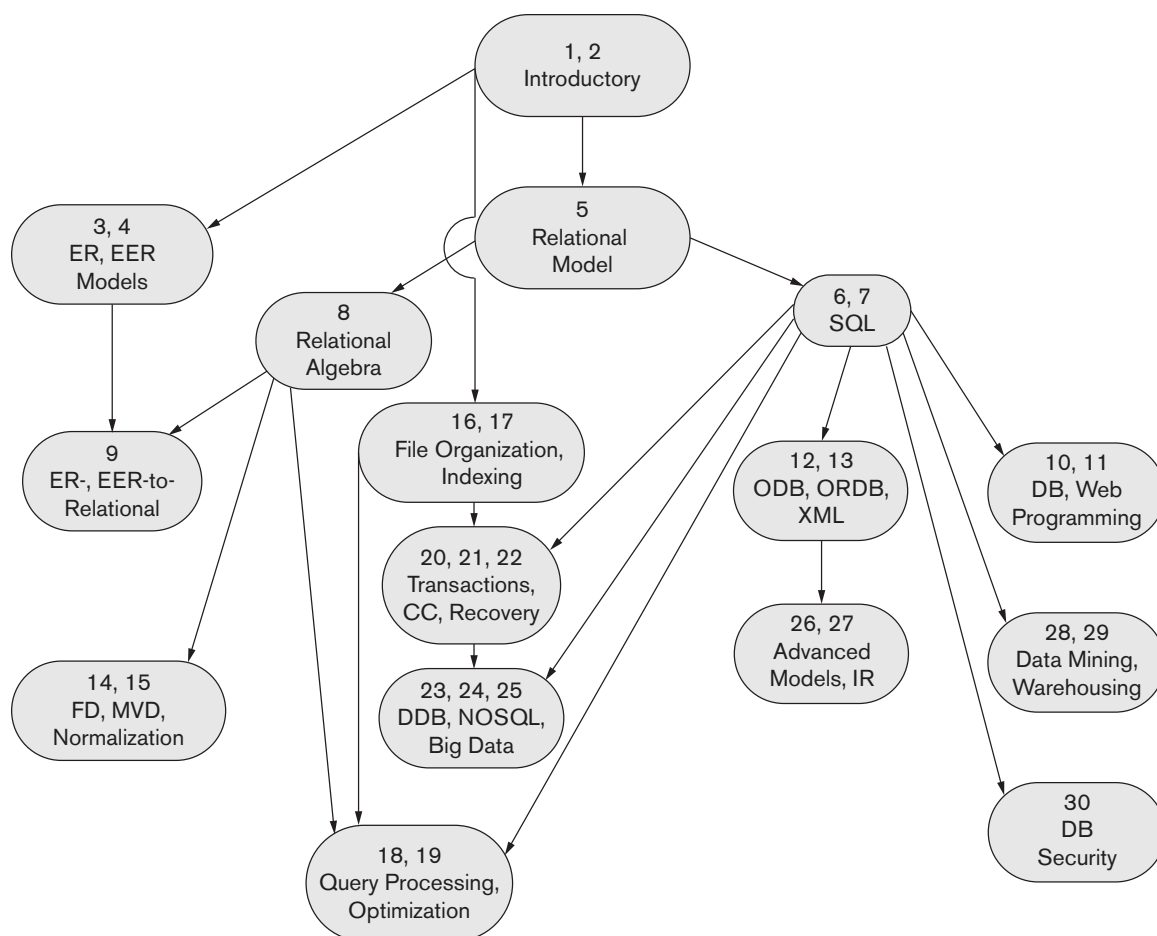
Supplemental Materials

Support material is available to qualified instructors at Pearson's instructor resource center (http://www.pearsonhighered.com/irc). For access, contact your local Pearson representative.

- PowerPoint lecture notes and figures.
- A solutions manual.

Acknowledgments

It is a great pleasure to acknowledge the assistance and contributions of many individuals to this effort. First, we would like to thank our editor, Matt Goldstein, for his guidance, encouragement, and support. We would like to acknowledge the excellent work of Rose Kernan for production management, Patricia Daly for a

thorough copy editing of the book, Martha McMaster for her diligence in proofing the pages, and Scott Disanno, Managing Editor of the production team. We also wish to thank Kelsey Loanes from Pearson for her continued help with the project, and reviewers Michael Doherty, Deborah Dunn, Imad Rahal, Karen Davis, Gilliean Lee, Leo Mark, Monisha Pulimood, Hassan Reza, Susan Vrbsky, Li Da Xu, Weining Zhang and Vincent Oria.

Ramez Elmasri would like to thank Kulsawasd Jitkajornwanich, Vivek Sharma, and Surya Swaminathan for their help with preparing some of the material in Chapter 24. Sham Navathe would like to acknowledge the following individuals who helped in critically reviewing and revising various topics. Dan Forsythe and Satish Damle for discussion of storage systems; Rafi Ahmed for detailed re-organization of the material on query processing and optimization; Harish Butani, Balaji Palanisamy, and Prajakta Kalmegh for their help with the Hadoop and MapReduce technology material; Vic Ghorpadey and Nenad Jukic for revision of the Data Warehousing material; and finally, Frank Rietta for newer techniques in database security, Kunal Malhotra for various discussions, and Saurav Sahay for advances in information retrieval systems.

We would like to repeat our thanks to those who have reviewed and contributed to previous editions of *Fundamentals of Database Systems*.

- **First edition.** Alan Apt (editor), Don Batory, Scott Downing, Dennis Heimbinger, Julia Hodges, Yannis Ioannidis, Jim Larson, Per-Ake Larson, Dennis McLeod, Rahul Patel, Nicholas Roussopoulos, David Stemple, Michael Stonebraker, Frank Tompa, and Kyu-Young Whang.

- **Second edition.** Dan Joraanstad (editor), Rafi Ahmed, Antonio Albano, David Beech, Jose Blakeley, Panos Chrysanthis, Suzanne Dietrich, Vic Ghorpadey, Goetz Graefe, Eric Hanson, Junguk L. Kim, Roger King, Vram Kouramajian, Vijay Kumar, John Lowther, Sanjay Manchanda, Toshimi Minoura, Inderpal Mumick, Ed Omiecinski, Girish Pathak, Raghu Ramakrishnan, Ed Robertson, Eugene Sheng, David Stotts, Marianne Winslett, and Stan Zdonick.

- **Third edition.** Maite Suarez-Rivas and Katherine Harutunian (editors); Suzanne Dietrich, Ed Omiecinski, Rafi Ahmed, Francois Bancilhon, Jose Blakeley, Rick Cattell, Ann Chervenak, David W. Embley, Henry A. Etlinger, Leonidas Fegaras, Dan Forsyth, Farshad Fotouhi, Michael Franklin, Sreejith Gopinath, Goetz Craefe, Richard Hull, Sushil Jajodia, Ramesh K. Karne, Harish Kotbagi, Vijay Kumar, Tarcisio Lima, Ramon A. Mata-Toledo, Jack McCaw, Dennis McLeod, Rokia Missaoui, Magdi Morsi, M. Narayanaswamy, Carlos Ordonez, Joan Peckham, Betty Salzberg, Ming-Chien Shan, Junping Sun, Rajshekhar Sunderraman, Aravindan Veerasamy, and Emilia E. Villareal.

- **Fourth edition.** Maite Suarez-Rivas, Katherine Harutunian, Daniel Rausch, and Juliet Silveri (editors); Phil Bernhard, Zhengxin Chen, Jan Chomicki, Hakan Ferhatosmanoglu, Len Fisk, William Hankley, Ali R. Hurson, Vijay Kumar, Peretz Shoval, Jason T. L. Wang (reviewers); Ed Omiecinski (who contributed to Chapter 27). Contributors from the University of Texas at

R. E.

S.B.N.

This page intentionally left blank

Contents

■ part 2

Conceptual Data Modeling and Database Design ■

■ part **3**

The Relational Data Model and SQL ■

chapter **9**   **Relational Database Design by ER- and EER-to-Relational Mapping   289**

■ **part 4**

Database Programming Techniques ■

chapter **10**   **Introduction to SQL Programming Techniques   309**

chapter **11**   **Web Database Programming Using PHP   343**

■ part 5

Object, Object-Relational, and XML: Concepts, Models, Languages, and Standards ■

chapter 12   Object and Object-Relational Databases   363

chapter 13   XML: Extensible Markup Language   425

■ part 6

Database Design Theory and Normalization ■

■ part 7

File Structures, Hashing, Indexing, and Physical Database Design ■

■ part 8

Query Processing and Optimization ■

■ part 9

Transaction Processing, Concurrency Control, and Recovery ■

■ part 10

Distributed Databases, NOSQL Systems, and Big Data ■

chapter 23 Distributed Database Concepts   841

chapter 24 NOSQL Databases and Big Data Storage Systems   883

chapter **28** Data Mining Concepts   **1069**

chapter **29** Overview of Data Warehousing and OLAP   **1101**

■ part **12**

Additional Database Topics: Security ■

chapter **30** Database Security   **1121**

appendix **A** **Alternative Diagrammatic Notations for ER
                Models   1163**

appendix **B** **Parameters of Disks   1167**

appendix **C** **Overview of the QBE Language   1171**

appendix **D** **Overview of the Hierarchical Data Model**
                (located on the Companion Website at
                http://www.pearsonhighered.com/elmasri)

appendix **E** **Overview of the Network Data Model**
                (located on the Companion Website at
                http://www.pearsonhighered.com/elmasri)

Selected Bibliography   1179

Index   1215

About the Authors

Ramez Elmasri is a professor and the associate chairperson of the Department of Computer Science and Engineering at the University of Texas at Arlington. He has over 140 refereed research publications, and has supervised 16 PhD students and over 100 MS students. His research has covered many areas of database management and big data, including conceptual modeling and data integration, query languages and indexing techniques, temporal and spatio-temporal databases, bio-informatics databases, data collection from sensor networks, and mining/analysis of spatial and spatio-temporal data. He has worked as a consultant to various companies, including Digital, Honeywell, Hewlett Packard, and Action Technologies, as well as consulting with law firms on patents. He was the Program Chair of the 1993 International Conference on Conceptual Modeling (ER conference) and program vice-chair of the 1994 IEEE International Conference on Data Engineering. He has served on the ER conference steering committee and has been on the program committees of many conferences. He has given several tutorials at the VLDB, ICDE, and ER conferences. He also co-authored the book "Operating Systems: A Spiral Approach" (McGraw-Hill, 2009) with Gil Carrick and David Levine. Elmasri is a recipient of the UTA College of Engineering Outstanding Teaching Award in 1999. He holds a BS degree in Engineering from Alexandria University, and MS and PhD degrees in Computer Science from Stanford University.

Shamkant B. Navathe is a professor and the founder of the database research group at the College of Computing, Georgia Institute of Technology, Atlanta. He has worked with IBM and Siemens in their research divisions and has been a consultant to various companies including Digital, Computer Corporation of America, Hewlett Packard, Equifax, and Persistent Systems. He was the General Co-chairman of the 1996 International VLDB (Very Large Data Base) conference in Bombay, India. He was also program co-chair of ACM SIGMOD 1985 International Conference and General Co-chair of the IFIP WG 2.6 Data Semantics Workshop in 1995. He has served on the VLDB foundation and has been on the steering committees of several conferences. He has been an associate editor of a number of journals including *ACM Computing Surveys*, and *IEEE Transactions on Knowledge and Data Engineering.* He also co-authored the book "Conceptual Design: An Entity Relationship Approach" (Addison Wesley, 1992) with Carlo Batini and Stefano Ceri. Navathe is a fellow of the Association for Computing Machinery (ACM) and recipient of the IEEE TCDE Computer Science, Engineering and Education Impact award in 2015. Navathe holds a PhD from the University of Michigan and has over 150 refereed publications in journals and conferences.

part **1**

Introduction to Databases

This page intentionally left blank

Databases and
Database Users

Databases and database systems are an essential component of life in modern society: most of us encounter several activities every day that involve some interaction with a database. For example, if we go to the bank to deposit or withdraw funds, if we make a hotel or airline reservation, if we access a computerized library catalog to search for a bibliographic item, or if we purchase something online—such as a book, toy, or computer—chances are that our activities will involve someone or some computer program accessing a database. Even purchasing items at a supermarket often automatically updates the database that holds the inventory of grocery items.

These interactions are examples of what we may call **traditional database applications**, in which most of the information that is stored and accessed is either textual or numeric. In the past few years, advances in technology have led to exciting new applications of database systems. The proliferation of social media Web sites, such as Facebook, Twitter, and Flickr, among many others, has required the creation of huge databases that store nontraditional data, such as posts, tweets, images, and video clips. New types of database systems, often referred to as **big data** storage systems, or **NOSQL systems**, have been created to manage data for social media applications. These types of systems are also used by companies such as Google, Amazon, and Yahoo, to manage the data required in their Web search engines, as well as to provide **cloud storage**, whereby users are provided with storage capabilities on the Web for managing all types of data including documents, programs, images, videos and emails. We will give an overview of these new types of database systems in Chapter 24.

We now mention some other applications of databases. The wide availability of photo and video technology on cellphones and other devices has made it possible to

store images, audio clips, and video streams digitally. These types of files are becoming an important component of **multimedia databases**. **Geographic information systems (GISs)** can store and analyze maps, weather data, and satellite images. **Data warehouses** and **online analytical processing (OLAP)** systems are used in many companies to extract and analyze useful business information from very large databases to support decision making. **Real-time** and **active database technology** is used to control industrial and manufacturing processes. And database **search techniques** are being applied to the World Wide Web to improve the search for information that is needed by users browsing the Internet.

To understand the fundamentals of database technology, however, we must start from the basics of traditional database applications. In Section 1.1 we start by defining a database, and then we explain other basic terms. In Section 1.2, we provide a simple UNIVERSITY database example to illustrate our discussion. Section 1.3 describes some of the main characteristics of database systems, and Sections 1.4 and 1.5 categorize the types of personnel whose jobs involve using and interacting with database systems. Sections 1.6, 1.7, and 1.8 offer a more thorough discussion of the various capabilities provided by database systems and discuss some typical database applications. Section 1.9 summarizes the chapter.

The reader who desires a quick introduction to database systems can study Sections 1.1 through 1.5, then skip or browse through Sections 1.6 through 1.8 and go on to Chapter 2.

1.1 Introduction

Databases and database technology have had a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, social media, engineering, medicine, genetics, law, education, and library science. The word *database* is so commonly used that we must begin by defining what a database is. Our initial definition is quite general.

A **database** is a collection of related data.[1] By **data**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. Nowadays, this data is typically stored in mobile phones, which have their own simple database software. This data can also be recorded in an indexed address book or stored on a hard drive, using a personal computer and software such as Microsoft Access or Excel. This collection of related data with an implicit meaning is a database.

The preceding definition of database is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to

---

[1]We will use the word *data* as both singular and plural, as is common in database literature; the context will determine whether it is singular or plural. In standard English, *data* is used for plural and *datum* for singular.

constitute a database. However, the common use of the term *database* is usually more restricted. A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the **miniworld** or the **universe of discourse (UoD)**. Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

In other words, a database has some source from which data is derived, some degree of interaction with events in the real world, and an audience that is actively interested in its contents. The end users of a database may perform business transactions (for example, a customer buys a camera) or events may happen (for example, an employee has a baby) that cause the information in the database to change. In order for a database to be accurate and reliable at all times, it must be a true reflection of the miniworld that it represents; therefore, changes must be reflected in the database as soon as possible.

A database can be of any size and complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the computerized catalog of a large library may contain half a million entries organized under different categories—by primary author's last name, by subject, by book title—with each category organized alphabetically. A database of even greater size and complexity would be maintained by a social media company such as Facebook, which has more than a billion users. The database has to maintain information on which users are related to one another as *friends,* the postings of each user, which users are allowed to see each posting, and a vast amount of other types of information needed for the correct operation of their Web site. For such Web sites, a large number of databases are needed to keep track of the constantly changing information required by the social media Web site.

An example of a large commercial database is Amazon.com. It contains data for over 60 million active users, and millions of books, CDs, videos, DVDs, games, electronics, apparel, and other items. The database occupies over 42 terabytes (a terabyte is 10^{12} bytes worth of storage) and is stored on hundreds of computers (called servers). Millions of visitors access Amazon.com each day and use the database to make purchases. The database is continually updated as new books and other items are added to the inventory, and stock quantities are updated as purchases are transacted.

A database may be generated and maintained manually or it may be computerized. For example, a library card catalog is a database that may be created and maintained manually. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a

database management system. Of course, we are only concerned with computerized databases in this text.

A **database management system (DBMS)** is a computerized system that enables users to create and maintain a database. The DBMS is a *general-purpose software system* that facilitates the processes of *defining, constructing, manipulating,* and *sharing* databases among various users and applications. **Defining** a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called **meta-data**. **Constructing** the database is the process of storing the data on some storage medium that is controlled by the DBMS. **Manipulating** a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. **Sharing** a database allows multiple users and programs to access the database simultaneously.

An **application program** accesses the database by sending queries or requests for data to the DBMS. A **query**[2] typically causes some data to be retrieved; a **transaction** may cause some data to be read and some data to be written into the database.

Other important functions provided by the DBMS include *protecting* the database and *maintaining* it over a long period of time. **Protection** includes *system protection* against hardware or software malfunction (or crashes) and *security protection* against unauthorized or malicious access. A typical large database may have a life cycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.

It is not absolutely necessary to use general-purpose DBMS software to implement a computerized database. It is possible to write a customized set of programs to create and maintain the database, in effect creating a *special-purpose* DBMS software for a specific application, such as airlines reservations. In either case—whether we use a general-purpose DBMS or not—a considerable amount of complex software is deployed. In fact, most DBMSs are very complex software systems.

To complete our initial definitions, we will call the database and DBMS software together a **database system**. Figure 1.1 illustrates some of the concepts we have discussed so far.

1.2 An Example

Let us consider a simple example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1.2 shows the database structure and a few sample data records. The database is organized as five files, each of which

---

[2]The term *query*, originally meaning a question or an inquiry, is sometimes loosely used for all types of interactions with databases, including modifying the data.

Users/Programmers
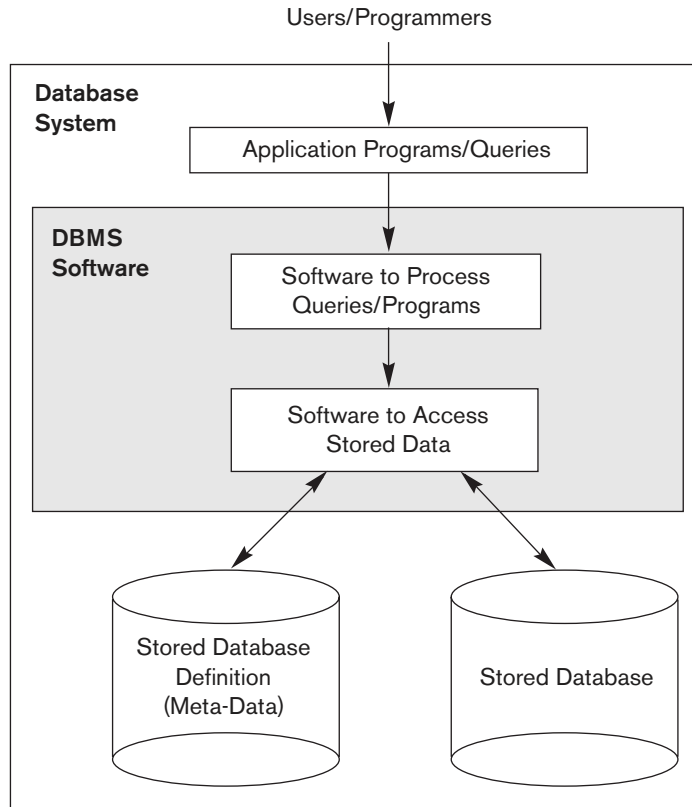
Figure 1.1
A simplified database
system environment.

stores **data records** of the same type.[3] The STUDENT file stores data on each student, the COURSE file stores data on each course, the SECTION file stores data on each section of a course, the GRADE_REPORT file stores the grades that students receive in the various sections they have completed, and the PREREQUISITE file stores the prerequisites of each course.

To *define* this database, we must specify the structure of the records of each file by specifying the different types of **data elements** to be stored in each record. In Figure 1.2, each STUDENT record includes data to represent the student's Name, Student_number, Class (such as freshman or '1', sophomore or '2', and so forth), and Major (such as mathematics or 'MATH' and computer science or 'CS'); each COURSE record includes data to represent the Course_name, Course_number, Credit_hours, and Department (the department that offers the course), and so on. We must also specify a **data type** for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, Student_number of STUDENT is an integer, and Grade of GRADE_REPORT is a

---

[3]We use the term *file* informally here. At a conceptual level, a *file* is a *collection* of records that may or may not be ordered.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores
student and course
information.

single character from the set {'A', 'B', 'C', 'D', 'F', 'I'}. We may also use a coding scheme to represent the values of a data item. For example, in Figure 1.2 we represent the Class of a STUDENT as 1 for freshman, 2 for sophomore, 3 for junior, 4 for senior, and 5 for graduate student.

To *construct* the UNIVERSITY database, we store data to represent each student, course, section, grade report, and prerequisite as a record in the appropriate file. Notice that records in the various files may be related. For example, the record for Smith in the STUDENT file is related to two records in the GRADE_REPORT file that specify Smith's grades in two sections. Similarly, each record in the PREREQUISITE file relates two course records: one representing the course and the other representing the prerequisite. Most medium-size and large databases include many types of records and have *many relationships* among the records.

Database *manipulation* involves querying and updating. Examples of queries are as follows:

- Retrieve the transcript—a list of all courses and grades—of 'Smith'
- List the names of students who took the section of the 'Database' course offered in fall 2008 and their grades in that section
- List the prerequisites of the 'Database' course

Examples of updates include the following:

- Change the class of 'Smith' to sophomore
- Create a new section for the 'Database' course for this semester
- Enter a grade of 'A' for 'Smith' in the 'Database' section of last semester

These informal queries and updates must be specified precisely in the query language of the DBMS before they can be processed.

At this stage, it is useful to describe the database as part of a larger undertaking known as an information system within an organization. The Information Technology (IT) department within an organization designs and maintains an information system consisting of various computers, storage systems, application software, and databases. Design of a new application for an existing database or design of a brand new database starts off with a phase called **requirements specification and analysis**. These requirements are documented in detail and transformed into a **conceptual design** that can be represented and manipulated using some computerized tools so that it can be easily maintained, modified, and transformed into a database implementation. (We will introduce a model called the Entity-Relationship model in Chapter 3 that is used for this purpose.) The design is then translated to a **logical design** that can be expressed in a data model implemented in a commercial DBMS. (Various types of DBMSs are discussed throughout the text, with an emphasis on relational DBMSs in Chapters 5 through 9.)

The final stage is **physical design**, during which further specifications are provided for storing and accessing the database. The database design is implemented, populated with actual data, and continuously maintained to reflect the state of the miniworld.

1.3 Characteristics of the Database Approach

A number of characteristics distinguish the database approach from the much older approach of writing customized programs to access data stored in files. In traditional **file processing**, each user defines and implements the files needed for a specific software application as part of programming the application. For example, one user, the *grade reporting office,* may keep files on students and their grades. Programs to print a student's transcript and to enter new grades are implemented as part of the application. A second user, the *accounting office*, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common up-to-date data.

In the database approach, a single repository maintains data that is defined once and then accessed by various users repeatedly through queries, transactions, and application programs. The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support of multiple views of the data
- Sharing of data and multiuser transaction processing

We describe each of these characteristics in a separate section. We will discuss additional characteristics of database systems in Sections 1.6 through 1.8.

1.3.1 Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the DBMS catalog, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data**, and it describes the structure of the primary database (Figure 1.1). It is important to note that some newer types of database systems, known as NOSQL systems, do not require meta-data. Rather the data is stored as **self-describing data** that includes the data item names and data values together in one structure (see Chapter 24).

The catalog is used by the DBMS software and also by database users who need information about the database structure. A general-purpose DBMS software package is not written for a specific database application. Therefore, it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access. The DBMS software must work equally well with *any number of database applications*—for example, a university database, a

banking database, or a company database—as long as the database definition is stored in the catalog.

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only *one specific database,* whose structure is declared in the application programs. For example, an application program written in C++ may have struct or class declarations. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and using these definitions.

For the example shown in Figure 1.2, the DBMS catalog will store the definitions of all the files shown. Figure 1.3 shows some entries in a database catalog. Whenever a request is made to access, say, the Name of a STUDENT record, the DBMS software refers to the catalog to determine the structure of the STUDENT file and the position and size of the Name data item within a STUDENT record. By contrast, in a typical file-processing application, the file structure and, in the extreme case, the exact location of Name within a STUDENT record are already coded within each program that accesses this data item.

RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
….	….	…..
….	….	…..
….	….	…..
Prerequisite_number	XXXXNNNN	PREREQUISITE

Figure 1.3
An example of a database catalog for the database in Figure 1.2.

Note: Major_type is defined as an enumerated type with all known majors.
XXXXNNNN is used to define a type with four alphabetic characters followed by four numeric digits.

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require *changing all programs* that access that file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence**.

For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure 1.4. If we want to add another piece of data to each STUDENT record, say the Birth_date, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we only need to change the *description* of STUDENT records in the catalog (Figure 1.3) to reflect the inclusion of the new data item Birth_date; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

In some types of database systems, such as object-oriented and object-relational systems (see Chapter 12), users can define operations on data as part of the database definitions. An **operation** (also called a *function* or *method*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence**.

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a **conceptual representation** of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

Looking at the example in Figures 1.2 and 1.3, the internal implementation of the STUDENT file may be defined by its record length—the number of characters (bytes) in each record—and each data item may be specified by its starting byte within a record and its length in bytes. The STUDENT record would thus be represented as shown in Figure 1.4. But a typical database user is not concerned with the location of each data item within a record or its length; rather, the user is concerned that when a reference is made to Name of STUDENT, the correct value is returned. A conceptual representation of the STUDENT records is shown in Figure 1.2. Many other details of file storage organization—such as the access paths specified on a

Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
Student_number	31	4
Class	35	1
Major	36	4

Figure 1.4
Internal storage format for a STUDENT record, based on the database catalog in Figure 1.3.

file—can be hidden from database users by the DBMS; we discuss storage details in Chapters 16 and 17.

In the database approach, the detailed structure and organization of each file are stored in the catalog. Database users and application programs refer to the conceptual representation of the files, and the DBMS extracts the details of file storage from the catalog when these are needed by the DBMS file access modules. Many data models can be used to provide this data abstraction to database users. A major part of this text is devoted to presenting various data models and the concepts they use to abstract the representation of data.

In object-oriented and object-relational databases, the abstraction process includes not only the data structure but also the operations on the data. These operations provide an abstraction of miniworld activities commonly understood by the users. For example, an operation CALCULATE_GPA can be applied to a STUDENT object to calculate the grade point average. Such operations can be invoked by the user queries or application programs without having to know the details of how the operations are implemented.

1.3.3 Support of Multiple Views of the Data

A database typically has many types of users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views. For example, one user of the database of Figure 1.2 may be interested only in accessing and printing the transcript of each student; the view for this user is shown in Figure 1.5(a). A second user, who is interested only in checking that students have taken all the prerequisites of each course for which the student registers, may require the view shown in Figure 1.5(b).

1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data

TRANSCRIPT

Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

(a)

COURSE_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

(b)

Figure 1.5
Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view.
(b) The COURSE_PREREQUISITES view.

do so in a controlled manner so that the result of the updates is correct. For example, when several reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger. These types of applications are generally called **online transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently.

The concept of a **transaction** has become central to many database applications. A transaction is an *executing program* or *process* that includes one or more database accesses, such as reading or updating of database records. Each transaction is supposed to execute a logically correct database access if executed in its entirety without interference from other transactions. The DBMS must enforce several transaction properties. The **isolation** property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The **atomicity** property ensures that either all the database operations in a transaction are executed or none are. We discuss transactions in detail in Part 9.

The preceding characteristics are important in distinguishing a DBMS from traditional file-processing software. In Section 1.6 we discuss additional features that characterize a DBMS. First, however, we categorize the different types of people who work in a database system environment.

1.4 Actors on the Scene

For a small personal database, such as the list of addresses discussed in Section 1.1, one person typically defines, constructs, and manipulates the database, and there is no sharing. However, in large organizations, many people are involved in the design, use, and maintenance of a large database with hundreds or thousands of users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the *actors on the scene.* In Section 1.5 we consider people who may be called *workers behind the scene*—those who work to maintain the database system environment but who are not actively interested in the database contents as part of their daily job.

1.4.1 Database Administrators

In any organization where many people use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA)**. The DBA is responsible for authorizing access to the database, coordinating and monitoring its use, and acquiring software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA is assisted by a staff that carries out these functions.

1.4.2 Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the responsibility of database designers to communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements. In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed. Database designers typically interact with each potential group of users and develop **views** of the database that meet the data and processing requirements of these groups. Each view is then analyzed and *integrated* with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

1.4.3 End Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query interface

to specify their requests and are typically middle- or high-level managers or other occasional browsers.

- **Naive** or **parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions**—that have been carefully programmed and tested. Many of these tasks are now available as **mobile apps** for use with mobile devices. The tasks that such users perform are varied. A few examples are:
  - Bank customers and tellers check account balances and post withdrawals and deposits.
  - Reservation agents or customers for airlines, hotels, and car rental companies check availability for a given request and make reservations.
  - Employees at receiving stations for shipping companies enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.
  - Social media users post and read items on social media Web sites.
- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.
- **Standalone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. An example is the user of a financial software package that stores a variety of personal financial data.

A typical DBMS provides multiple facilities to access a database. Naive end users need to learn very little about the facilities provided by the DBMS; they simply have to understand the user interfaces of the mobile apps or standard transactions designed and implemented for their use. Casual users learn only a few facilities that they may use repeatedly. Sophisticated users try to learn most of the DBMS facilities in order to achieve their complex requirements. Standalone users typically become very proficient in using a specific software package.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for standard canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers—commonly referred to as **software developers** or **software engineers**—should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

1.5 Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system environment.* These persons are typically not interested in the database content itself. We call them the *workers behind the scene*, and they include the following categories:

- **DBMS system designers and implementers** design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or **modules**, including modules for implementing the catalog, query language processing, interface processing, accessing and buffering data, controlling concurrency, and handling data recovery and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.

- **Tool developers** design and implement **tools**—the software packages that facilitate database modeling and design, database system design, and improved performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation. In many cases, independent software vendors develop and market these tools.

- **Operators and maintenance personnel** (system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Although these categories of workers behind the scene are instrumental in making the database system available to end users, they typically do not use the database contents for their own purposes.

1.6 Advantages of Using the DBMS Approach

In this section we discuss some additional advantages of using a DBMS and the capabilities that a good DBMS should possess. These capabilities are in addition to the four main characteristics discussed in Section 1.3. The DBA must utilize these capabilities to accomplish a variety of objectives related to the design, administration, and use of a large multiuser database.

1.6.1 Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example of Section 1.2; here, two groups of users might be the course registration personnel and the accounting office. In the traditional approach, each group independently keeps files on students. The

accounting office keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Other groups may further duplicate some or all of the same data in their own files.

This **redundancy** in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update—such as entering data on a new student—multiple times: once for each file where student data is recorded. This leads to *duplication of effort.* Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases. Third, files that represent the same data may become *inconsistent.* This may happen because an update is applied to some of the files but not to others. Even if an update—such as adding a new student—is applied to all the appropriate files, the data concerning the student may still be *inconsistent* because the updates are applied independently by each user group. For example, one user group may enter a student's birth date erroneously as 'JAN-19-1988', whereas the other user groups may enter the correct value of 'JAN-29-1988'.

In the database approach, the views of different user groups are integrated during database design. Ideally, we should have a database design that stores each logical data item—such as a student's name or birth date—in *only one place* in the database. This is known as **data normalization**, and it ensures consistency and saves storage space (data normalization is described in Part 6 of the text).

However, in practice, it is sometimes necessary to use **controlled redundancy** to improve the performance of queries. For example, we may store Student_name and Course_number redundantly in a GRADE_REPORT file (Figure 1.6(a)) because whenever we retrieve a GRADE_REPORT record, we want to retrieve the student name and course number along with the grade, student number, and section identifier. By placing all the data together, we do not have to search multiple files to collect this data. This is known as **denormalization**. In such cases, the DBMS should

Figure 1.6
Redundant storage of Student_name and Course_name in GRADE_REPORT. (a) Consistent data. (b) Inconsistent record.

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Smith	112	MATH2410	B
17	Smith	119	CS1310	C
8	Brown	85	MATH2410	A
8	Brown	92	CS1310	A
8	Brown	102	CS3320	B
8	Brown	135	CS3380	A

(a)

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Brown	112	MATH2410	B

(b)

have the capability to *control* this redundancy in order to prohibit inconsistencies among the files. This may be done by automatically checking that the Student_name–Student_number values in any GRADE_REPORT record in Figure 1.6(a) match one of the Name–Student_number values of a STUDENT record (Figure 1.2). Similarly, the Section_identifier–Course_number values in GRADE_REPORT can be checked against SECTION records. Such checks can be specified to the DBMS during database design and automatically enforced by the DBMS whenever the GRADE_REPORT file is updated. Figure 1.6(b) shows a GRADE_REPORT record that is inconsistent with the STUDENT file in Figure 1.2; this kind of error may be entered if the redundancy is *not controlled.* Can you tell which part is inconsistent?

1.6.2 Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database. For example, financial data such as salaries and bonuses is often considered confidential, and only authorized persons are allowed to access such data. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and update. Hence, the type of access operation—retrieval or update—must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS should provide a **security and authorization subsystem**, which the DBA uses to create accounts and to specify account restrictions. Then, the DBMS should enforce these restrictions automatically. Notice that we can apply similar controls to the DBMS software. For example, only the DBA's staff may be allowed to use certain **privileged software**, such as the software for creating new accounts. Similarly, parametric users may be allowed to access the database only through the predefined apps or canned transactions developed for their use. We discuss database security and authorization in Chapter 30.

1.6.3 Providing Persistent Storage for Program Objects

Databases can be used to provide **persistent storage** for program objects and data structures. This is one of the main reasons for **object-oriented database systems** (see Chapter 12). Programming languages typically have complex data structures, such as structs or class definitions in C++ or Java. The values of program variables or objects are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable or object structure. Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be **persistent**, since it survives the termination of program execution and can later be directly retrieved by another program.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called **impedance mismatch problem**, since the data structures provided by the DBMS were incompatible with the programming language's data structures. Object-oriented database systems typically offer data structure **compatibility** with one or more object-oriented programming languages.

1.6.4 Providing Storage Structures and Search Techniques for Efficient Query Processing

Database systems must provide capabilities for *efficiently executing queries and updates.* Because the database is typically stored on disk, the DBMS must provide specialized data structures and search techniques to speed up disk search for the desired records. Auxiliary files called **indexes** are often used for this purpose. Indexes are typically based on tree data structures or hash data structures that are suitably modified for disk search. In order to process the database records needed by a particular query, those records must be copied from disk to main memory. Therefore, the DBMS often has a **buffering** or **caching** module that maintains parts of the database in main memory buffers. In general, the operating system is responsible for disk-to-memory buffering. However, because data buffering is crucial to the DBMS performance, most DBMSs do their own data buffering.

The **query processing and optimization** module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures. The choice of which indexes to create and maintain is part of *physical database design and tuning,* which is one of the responsibilities of the DBA staff. We discuss query processing and optimization in Part 8 of the text.

1.6.5 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update transaction, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Disk backup is also necessary in case of a catastrophic disk failure. We discuss recovery and backup in Chapter 22.

1.6.6 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include apps for mobile users, query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for standalone users. Both forms-style interfaces and menu-driven interfaces are commonly known as

graphical user interfaces (GUIs). Many specialized languages and environments exist for specifying GUIs. Capabilities for providing Web GUI interfaces to a database—or Web-enabling a database—are also quite common.

1.6.7 Representing Complex Relationships among Data

A database may include numerous varieties of data that are interrelated in many ways. Consider the example shown in Figure 1.2. The record for 'Brown' in the STUDENT file is related to four records in the GRADE_REPORT file. Similarly, each section record is related to one course record and to a number of GRADE_REPORT records—one for each student who completed that section. A DBMS must have the capability to represent a variety of complex relationships among the data, to define new relationships as they arise, and to retrieve and update related data easily and efficiently.

1.6.8 Enforcing Integrity Constraints

Most database applications have certain **integrity constraints** that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item. For example, in Figure 1.3, we specified that the value of the Class data item within each STUDENT record must be a one-digit integer and that the value of Name must be a string of no more than 30 alphabetic characters. To restrict the value of Class between 1 and 5 would be an additional constraint that is not shown in the current catalog. A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files. For example, in Figure 1.2, we can specify that *every section record must be related to a course record*. This is known as a **referential integrity** constraint. Another type of constraint specifies uniqueness on data item values, such as *every course record must have a unique value for Course_number*. This is known as a **key** or **uniqueness** constraint. These constraints are derived from the meaning or **semantics** of the data and of the miniworld it represents. It is the responsibility of the database designers to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry. For typical large applications, it is customary to call such constraints **business rules**.

A data item may be entered erroneously and still satisfy the specified integrity constraints. For example, if a student receives a grade of 'A' but a grade of 'C' is entered in the database, the DBMS *cannot* discover this error automatically because 'C' is a valid value for the Grade data type. Such data entry errors can only be discovered manually (when the student receives the grade and complains) and corrected later by updating the database. However, a grade of 'Z' would be rejected automatically by the DBMS because 'Z' is not a valid value for the Grade data type. When we discuss each data model in subsequent chapters, we will introduce rules that pertain to

that model implicitly. For example, in the Entity-Relationship model in Chapter 3, a relationship must involve at least two entities. Rules that pertain to a specific data model are called **inherent rules** of the data model.

1.6.9 Permitting Inferencing and Actions Using Rules and Triggers

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems**. For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as **rules**, which when compiled and maintained by the DBMS can determine all students on probation. In a traditional DBMS, an explicit *procedural program code* would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. In today's relational database systems, it is possible to associate **triggers** with tables. A trigger is a form of a rule activated by updates to the table, which results in performing some additional operations to some other tables, sending messages, and so on. More involved procedures to enforce rules are popularly called **stored procedures**; they become a part of the overall database definition and are invoked appropriately when certain conditions are met. More powerful functionality is provided by **active database systems**, which provide active rules that can automatically initiate actions when certain events and conditions occur (see Chapter 26 for introductions to active databases in Section 26.1 and deductive databases in Section 26.5).

1.6.10 Additional Implications of Using the Database Approach

This section discusses a few additional implications of using the database approach that can benefit most organizations.

Potential for Enforcing Standards. The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on. The DBA can enforce standards in a centralized database environment more easily than in an environment where each user group has control of its own data files and software.

Reduced Application Development Time. A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. Designing and implementing a large multiuser database from scratch may take more time than writing a single specialized file application. However, once a database is up and running, substantially less time is generally required to create new applications

using DBMS facilities. Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a file system.

Flexibility. It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs information not currently in the database. In response, it may be necessary to add a file to the database or to extend the data elements in an existing file. Modern DBMSs allow certain types of evolutionary changes to the structure of the database without affecting the stored data and the existing application programs.

Availability of Up-to-Date Information. A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

Economies of Scale. The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments as well as redundancies among applications. This enables the whole organization to invest in more powerful processors, storage devices, or networking gear, rather than having each department purchase its own (lower performance) equipment. This reduces overall costs of operation and management.

1.7 A Brief History of Database Applications

We now give a brief historical overview of the applications that use DBMSs and how these applications provided the impetus for new types of database systems.

1.7.1 Early Database Applications Using Hierarchical and Network Systems

Many early database applications maintained records in large organizations such as corporations, universities, hospitals, and banks. In many of these applications, there were large numbers of records of similar structure. For example, in a university application, similar information would be kept for each student, each course, each grade record, and so on. There were also many types of records and many interrelationships among them.

One of the main problems with early database systems was the intermixing of conceptual relationships with the physical storage and placement of records on disk. Hence, these systems did not provide sufficient *data abstraction* and *program-data independence* capabilities. For example, the grade records of a particular student could be physically stored next to the student record. Although this provided very

efficient access for the original queries and transactions that the database was designed to handle, it did not provide enough flexibility to access records efficiently when new queries and transactions were identified. In particular, new queries that required a different storage organization for efficient processing were quite difficult to implement efficiently. It was also laborious to reorganize the database when changes were made to the application's requirements.

Another shortcoming of early systems was that they provided only programming language interfaces. This made it time-consuming and expensive to implement new queries and transactions, since new programs had to be written, tested, and debugged. Most of these database systems were implemented on large and expensive mainframe computers starting in the mid-1960s and continuing through the 1970s and 1980s. The main types of early systems were based on three main paradigms: hierarchical systems, network model–based systems, and inverted file systems.

1.7.2 Providing Data Abstraction and Application Flexibility with Relational Databases

Relational databases were originally proposed to separate the physical storage of data from its conceptual representation and to provide a mathematical foundation for data representation and querying. The relational data model also introduced high-level query languages that provided an alternative to programming language interfaces, making it much faster to write new queries. Relational representation of data somewhat resembles the example we presented in Figure 1.2. Relational systems were initially targeted to the same applications as earlier systems, and provided flexibility to develop new queries quickly and to reorganize the database as requirements changed. Hence, *data abstraction* and *program-data independence* were much improved when compared to earlier systems.

Early experimental relational systems developed in the late 1970s and the commercial relational database management systems (RDBMS) introduced in the early 1980s were quite slow, since they did not use physical storage pointers or record placement to access related data records. With the development of new storage and indexing techniques and better query processing and optimization, their performance improved. Eventually, relational databases became the dominant type of database system for traditional database applications. Relational databases now exist on almost all types of computers, from small personal computers to large servers.

1.7.3 Object-Oriented Applications and the Need for More Complex Databases

The emergence of object-oriented programming languages in the 1980s and the need to store and share complex, structured objects led to the development of object-oriented databases (OODBs). Initially, OODBs were considered a competitor

to relational databases, since they provided more general data structures. They also incorporated many of the useful object-oriented paradigms, such as abstract data types, encapsulation of operations, inheritance, and object identity. However, the complexity of the model and the lack of an early standard contributed to their limited use. They are now mainly used in specialized applications, such as engineering design, multimedia publishing, and manufacturing systems. Despite expectations that they will make a big impact, their overall penetration into the database products market remains low. In addition, many object-oriented concepts were incorporated into the newer versions of relational DBMSs, leading to object-relational database management systems, known as ORDBMSs.

1.7.4 Interchanging Data on the Web for E-Commerce Using XML

The World Wide Web provides a large network of interconnected computers. Users can create static Web pages using a Web publishing language, such as Hyper-Text Markup Language (HTML), and store these documents on Web servers where other users (clients) can access them and view them through Web browsers. Documents can be linked through **hyperlinks**, which are pointers to other documents. Starting in the 1990s, electronic commerce (e-commerce) emerged as a major application on the Web. Much of the critical information on e-commerce Web pages is dynamically extracted data from DBMSs, such as flight information, product prices, and product availability. A variety of techniques were developed to allow the interchange of dynamically extracted data on the Web for display on Web pages. The eXtended Markup Language (XML) is one standard for interchanging data among various types of databases and Web pages. XML combines concepts from the models used in document systems with database modeling concepts. Chapter 13 is devoted to an overview of XML.

1.7.5 Extending Database Capabilities for New Applications

The success of database systems in traditional applications encouraged developers of other types of applications to attempt to use them. Such applications traditionally used their own specialized software and file and data structures. Database systems now offer extensions to better support the specialized requirements for some of these applications. The following are some examples of these applications:

- **Scientific** applications that store large amounts of data resulting from scientific experiments in areas such as high-energy physics, the mapping of the human genome, and the discovery of protein structures

- Storage and retrieval of **images**, including scanned news or personal photographs, satellite photographic images, and images from medical procedures such as x-rays and MRI (magnetic resonance imaging) tests

- Storage and retrieval of **videos**, such as movies, and **video clips** from news or personal digital cameras
- **Data mining** applications that analyze large amounts of data to search for the occurrences of specific patterns or relationships, and for identifying unusual patterns in areas such as credit card fraud detection
- **Spatial** applications that store and analyze spatial locations of data, such as weather information, maps used in geographical information systems, and automobile navigational systems
- **Time series** applications that store information such as economic data at regular points in time, such as daily sales and monthly gross national product figures

It was quickly apparent that basic relational systems were not very suitable for many of these applications, usually for one or more of the following reasons:

- More complex data structures were needed for modeling the application than the simple relational representation.
- New data types were needed in addition to the basic numeric and character string types.
- New operations and query language constructs were necessary to manipulate the new data types.
- New storage and indexing structures were needed for efficient searching on the new data types.

This led DBMS developers to add functionality to their systems. Some functionality was general purpose, such as incorporating concepts from object-oriented databases into relational systems. Other functionality was special purpose, in the form of optional modules that could be used for specific applications. For example, users could buy a time series module to use with their relational DBMS for their time series application.

1.7.6 Emergence of Big Data Storage Systems and NOSQL Databases

In the first decade of the twenty-first century, the proliferation of applications and platforms such as social media Web sites, large e-commerce companies, Web search indexes, and cloud storage/backup led to a surge in the amount of data stored on large databases and massive servers. New types of database systems were necessary to manage these huge databases—systems that would provide fast search and retrieval as well as reliable and safe storage of nontraditional types of data, such as social media posts and tweets. Some of the requirements of these new systems were not compatible with SQL relational DBMSs (SQL is the standard data model and language for relational databases). The term *NOSQL* is generally interpreted as Not Only SQL, meaning that in systems than manage large amounts of data, some of the data is stored using SQL systems, whereas other data would be stored using NOSQL, depending on the application requirements.

1.8 **When Not to Use a DBMS**

In spite of the advantages of using a DBMS, there are a few situations in which a DBMS may involve unnecessary overhead costs that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:

- High initial investment in hardware, software, and training
- The generality that a DBMS provides for defining and processing data
- Overhead for providing security, concurrency control, recovery, and integrity functions

Therefore, it may be more desirable to develop customized database applications under the following circumstances:

- Simple, well-defined database applications that are not expected to change at all
- Stringent, real-time requirements for some application programs that may not be met because of DBMS overhead
- Embedded systems with limited storage capacity, where a general-purpose DBMS would not fit
- No multiple-user access to data

Certain industries and applications have elected not to use general-purpose DBMSs. For example, many computer-aided design (CAD) tools used by mechanical and civil engineers have proprietary file and data management software that is geared for the internal manipulations of drawings and 3D objects. Similarly, communication and switching systems designed by companies like AT&T were early manifestations of database software that was made to run very fast with hierarchically organized data for quick access and routing of calls. GIS implementations often implement their own data organization schemes for efficiently implementing functions related to processing maps, physical contours, lines, polygons, and so on.

1.9 **Summary**

In this chapter we defined a database as a collection of related data, where *data* means recorded facts. A typical database represents some aspect of the real world and is used for specific purposes by one or more groups of users. A DBMS is a generalized software package for implementing and maintaining a computerized database. The database and software together form a database system. We identified several characteristics that distinguish the database approach from traditional file-processing applications, and we discussed the main categories of database users, or the *actors on the scene*. We noted that in addition to database users, there are several categories of support personnel, or *workers behind the scene*, in a database environment.

We presented a list of capabilities that should be provided by the DBMS software to the DBA, database designers, and end users to help them design, administer, and use a database. Then we gave a brief historical perspective on the evolution of database applications. We pointed out the recent rapid growth of the amounts and types of data that must be stored in databases, and we discussed the emergence of new systems for handling "big data" applications. Finally, we discussed the overhead costs of using a DBMS and discussed some situations in which it may not be advantageous to use one.

Review Questions

1.1. Define the following terms: *data*, *database*, *DBMS*, *database system*, *database catalog*, *program-data independence*, *user view*, *DBA*, *end user*, *canned transaction*, *deductive database system*, *persistent object*, *meta-data*, and *transaction-processing application*.

1.2. What four main types of actions involve databases? Briefly discuss each.

1.3. Discuss the main characteristics of the database approach and how it differs from traditional file systems.

1.4. What are the responsibilities of the DBA and the database designers?

1.5. What are the different types of database end users? Discuss the main activities of each.

1.6. Discuss the capabilities that should be provided by a DBMS.

1.7. Discuss the differences between database systems and information retrieval systems.

Exercises

1.8. Identify some informal queries and update operations that you would expect to apply to the database shown in Figure 1.2.

1.9. What is the difference between controlled and uncontrolled redundancy? Illustrate with examples.

1.10. Specify all the relationships among the records of the database shown in Figure 1.2.

1.11. Give some additional views that may be needed by other user groups for the database shown in Figure 1.2.

1.12. Cite some examples of integrity constraints that you think can apply to the database shown in Figure 1.2.

1.13. Give examples of systems in which it may make sense to use traditional file processing instead of a database approach.

1.14. Consider Figure 1.2.

    a. If the name of the 'CS' (Computer Science) Department changes to 'CSSE' (Computer Science and Software Engineering) Department and the corresponding prefix for the course number also changes, identify the columns in the database that would need to be updated.

    b. Can you restructure the columns in the COURSE, SECTION, and PREREQUISITE tables so that only one column will need to be updated?

Selected Bibliography

The October 1991 issue of *Communications of the ACM* and Kim (1995) include several articles describing next-generation DBMSs; many of the database features discussed in the former are now commercially available. The March 1976 issue of *ACM Computing Surveys* offers an early introduction to database systems and may provide a historical perspective for the interested reader. We will include references to other concepts, systems, and applications introduced in this chapter in the later text chapters that discuss each topic in more detail.

This page intentionally left blank

Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with a client/server system architecture. The recent growth in the amount of data requiring storage has led to database systems with distributed architectures comprised of thousands of computers that manage the data stores. This evolution mirrors the trends in computing, where large centralized mainframe computers are replaced by hundreds of distributed workstations and personal computers connected via communications networks to various types of server machines—Web servers, database servers, file servers, application servers, and so on. The current **cloud computing** environments consist of thousands of large servers managing so-called **big data** for users on the Web.

In a basic client/server DBMS architecture, the system functionality is distributed between two types of modules.[1] A **client module** is typically designed so that it will run on a mobile device, user workstation, or personal computer (PC). Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as apps for mobile devices, or forms- or menu-based GUIs (graphical user interfaces) for PCs. The other kind of module, called a **server module**, typically handles data storage, access, search, and other functions. We discuss client/server architectures in more detail in Section 2.5. First, we must study more basic concepts that will give us a better understanding of modern database architectures.

---

[1]As we shall see in Section 2.5, there are variations on this simple *two-tier* client/server architecture.

In this chapter we present the terminology and basic concepts that will be used throughout the text. Section 2.1 discusses data models and defines the concepts of schemas and instances, which are fundamental to the study of database systems. We discuss the three-schema DBMS architecture and data independence in Section 2.2; this provides a user's perspective on what a DBMS is supposed to do. In Section 2.3 we describe the types of interfaces and languages that are typically provided by a DBMS. Section 2.4 discusses the database system software environment. Section 2.5 gives an overview of various types of client/server architectures. Finally, Section 2.6 presents a classification of the types of DBMS packages. Section 2.7 summarizes the chapter.

The material in Sections 2.4 through 2.6 provides detailed concepts that may be considered as supplementary to the basic introductory material.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. **Data abstraction** generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.[2] By *structure of a database* we mean the data types, relationships, and constraints that apply to the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the **dynamic aspect** or **behavior** of a database application. This allows the database designer to specify a set of valid user-defined operations that are allowed on the database objects.[3] An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the *basic data model operations*. Concepts to specify behavior are fundamental to object-oriented data models (see Chapter 12) but are also being incorporated in more traditional data models. For example, object-relational models (see Chapter 12) extend the basic relational model to include such concepts, among others. In the basic relational data model, there is a provision to attach behavior to the relations in the form of persistent stored modules, popularly known as stored procedures (see Chapter 10).

---

[2]Sometimes the word *model* is used to denote a specific database description, or schema—for example, *the marketing data model*. We will not use this interpretation.

[3]The inclusion of concepts to describe behavior reflects a trend whereby database design and software design activities are increasingly being combined into a single activity. Traditionally, specifying behavior is associated with software design.

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks. Concepts provided by physical data models are generally meant for computer specialists, not for end users. Between these two extremes is a class of **representational** (or **implementation**) **data models**,[4] which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project from the miniworld that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project. Chapter 3 presents the **entity–relationship model**—a popular high-level conceptual data model. Chapter 4 describes additional abstractions used for advanced modeling, such as generalization, specialization, and categories (union types).

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Part 3 of the text is devoted to the relational data model, and its constraints, operations, and languages.[5] The SQL standard for relational databases is described in Chapters 6 and 7. Representational data models represent data by using record structures and hence are sometimes called **record-based data models**.

We can regard the **object data model** as an example of a new family of higher-level implementation data models that are closer to conceptual data models. A standard for object databases called the ODMG object model has been proposed by the Object Data Management Group (ODMG). We describe the general characteristics of object databases and the object model proposed standard in Chapter 12. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An

---

[4]The term *implementation data model* is not a standard term; we have introduced it to refer to the available data models in commercial database systems.

[5]A summary of the hierarchical and network data models is included in Appendices D and E. They are accessible from the book's Web site.

access path is a search structure that makes the search for particular database records efficient, such as indexing or hashing. We discuss physical storage techniques and access structures in Chapters 16 and 17. An **index** is an example of an access path that allows direct access to data using an index term or a keyword. It is similar to the index at the end of this text, except that it may be organized in a linear, hierarchical (tree-structured), or some other fashion.

Another class of data models is known as **self-describing data models**. The data storage in systems based on these models combines the description of the data with the data values themselves. In traditional DBMSs, the description (schema) is separated from the data. These models include **XML** (see Chapter 12) as well as many of the **key-value stores** and **NOSQL systems** (see Chapter 24) that were recently created for managing big data.

2.1.2 Schemas, Instances, and Database State

In a data model, it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently.[6] Most data models have certain conventions for displaying schemas as diagrams.[7] A displayed schema is called a **schema diagram**. Figure 2.1 shows a schema diagram for the database shown in Figure 1.2; the diagram displays the structure of each record type but not the actual instances of records.

Figure 2.1
Schema diagram for the database in Figure 1.2.

STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

PREREQUISITE

Course_number	Prerequisite_number

SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

---

[6]Schema changes are usually needed as the requirements of the database applications change. Most database systems include operations for allowing schema changes.

[7]It is customary in database parlance to use *schemas* as the plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is also sometimes used to refer to a schema.

We call each object in the schema—such as STUDENT or COURSE—a **schema construct**.

A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example, Figure 2.1 shows neither the data type of each data item nor the relationships among the various files. Many types of constraints are not represented in schema diagrams. A constraint such as *students majoring in computer science must take CS1310 before the end of their sophomore year* is quite difficult to represent diagrammatically.

The actual data in a database may change quite frequently. For example, the database shown in Figure 1.2 changes every time we add a new student or enter a new grade. The data in the database at a particular moment in time is called a **database state** or **snapshot**. It is also called the *current* set of **occurrences** or **instances** in the database. In a given database state, each schema construct has its own *current set* of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.[8] The DBMS is partly responsible for ensuring that every state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important and the schema must be designed with utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the Date_of_birth to the STUDENT schema in Figure 2.1. This is known as **schema evolution**. Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

---

[8]The current state is also called the *current snapshot* of the database. It has also been called a *database instance*, but we prefer to use the term *instance* to refer to individual records.

2.2 Three-Schema Architecture and Data Independence

Three of the four important characteristics of the database approach, listed in Section 1.3, are (1) use of a catalog to store the database description (schema) so as to make it self-describing, (2) insulation of programs and data (program-data and program-operation independence), and (3) support of multiple user views. In this section we specify an architecture for database systems, called the **three-schema architecture**,[9] that was proposed to help achieve and visualize these characteristics. Then we discuss further the concept of data independence.

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications from the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

Figure 2.2
The three-schema architecture.
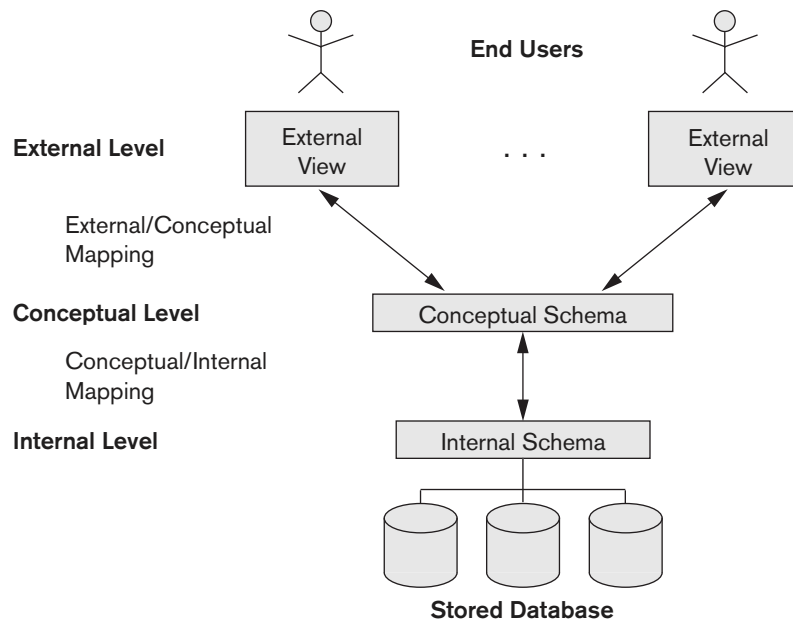
9This is also known as the ANSI/SPARC (American National Standards Institute/ Standards Planning And Requirements Committee) architecture, after the committee that proposed it (Tsichritzis & Klug, 1978).

2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented. This *implementation conceptual schema* is often based on a *conceptual schema design* in a high-level data model.

3. The **external** or **view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. As in the previous level, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level conceptual data model.

The three-schema architecture is a convenient tool with which the user can visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely and explicitly, but they support the three-schema architecture to some extent. Some older DBMSs may include physical-level details in the conceptual schema. The three-level ANSI architecture has an important place in database technology development because it clearly separates the users' external level, the database's conceptual level, and the internal storage level for designing a database. It is very much applicable in the design of DBMSs, even today. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information (for example, a relational DBMS like Oracle or SQLServer uses SQL for this).

Notice that the three schemas are only *descriptions* of data; the actual data is stored at the physical level only. In the three-schema architecture, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings**. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, it is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

The three-schema architecture can be used to further explain the concept of **data independence**, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We

may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.5(a) should not be affected by changing the GRADE_REPORT file (or record type) shown in Figure 1.2 into the one shown in Figure 1.6(a). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of SECTION records (Figure 1.2) by semester and year should not require a query such as *list all sections offered in fall 2008* to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Generally, physical data independence exists in most databases and file environments where physical details, such as the exact location of data on disk, and hardware details of storage encoding, placement, compression, splitting, merging of records, and so on are hidden from the user. Applications remain unaware of these details. On the other hand, logical data independence is harder to achieve because it allows structural and constraint changes without affecting application programs—a much stricter requirement.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

2.3 Database Languages and Interfaces

In Section 1.4 we discussed the variety of users supported by a DBMS. The DBMS must provide appropriate languages and interfaces for each category of users. In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

2.3.1 DBMS Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first step is to specify conceptual and internal schemas for the database and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language** (**DDL**), is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language** (**SDL**), is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. In most relational DBMSs today, there *is no specific language* that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage of files. These permit the DBA staff to control indexing choices and mapping of data to storage. For a true three-schema architecture, we would need a third language, the **view definition language** (**VDL**), to specify user views and their mappings to the conceptual schema, but in most DBMSs *the DDL is used to define both conceptual and external schemas*. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries (see Chapters 6 and 7).

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language** (**DML**) for these purposes.

In current DBMSs, the preceding types of languages are usually *not considered distinct languages*; rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, which is usually done by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language (see Chapters 6 and 7), which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification, schema evolution, and many other features. The SDL was a component in early versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A **high-level** or **nonprocedural** DML can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so

that they can be extracted by a precompiler and processed by the DBMS. A **low-level** or **procedural** DML *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Therefore, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement; therefore, they are called **set-at-a-time** or **set-oriented** DMLs. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; therefore, such languages are also called **declarative**.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**.[10] On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.[11]

Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are **user-friendly interfaces** for interacting with the database; these can also be used by casual users or others who do not want to learn the details of a high-level query language. We discuss these types of interfaces next.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

Menu-based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options (called **menus)** that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step-by-step by picking options from a menu that is displayed by the system. Pull-down menus are a very popular technique in **Web-based user interfaces**. They are also often used in **browsing interfaces**, which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Apps for Mobile Devices. These interfaces present mobile users with access to their data. For example, banking, reservations, and insurance companies, among many others, provide apps that allow users to access their data through a mobile phone or mobile device. The apps have built-in programmed interfaces that typically

---

[10]In object databases, the host and data sublanguages typically form one integrated language—for example, C++ with some extensions to support database functionality. Some relational systems also provide integrated languages—for example, Oracle's PL/SQL.

[11]According to the English meaning of the word *query*, it should really be used to describe retrievals only, not updates.

allow users to login using their account name and password; the apps then provide a limited menu of options for mobile access to the user data, as well as options such as paying bills (for banks) or making reservations (for reservation Web sites).

Forms-based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the **form** entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages**, which are special languages that help programmers specify such forms. SQL*Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema. Oracle Forms is a component of the Oracle product suite that provides an extensive set of features to design and build applications using forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

Graphical User Interfaces. A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms.

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to *understand* them. A natural language interface usually has its own *schema*, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, that are used to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

Keyword-based Database Search. These are somewhat similar to Web search engines, which accept strings of natural language (like English or Spanish) words and match them with documents at specific sites (for local search engines) or Web pages on the Web at large (for engines like Google or Ask). They use predefined indexes on words and use ranking functions to retrieve and present resulting documents in a decreasing degree of match. Such "free form" textual query interfaces are not yet common in structured relational databases, although a research area called **keyword-based querying** has emerged recently for relational databases.

Speech Input and Output. Limited use of speech as an input query and speech as an answer to a question or result of a request is becoming commonplace. Applications with limited vocabularies, such as inquiries for telephone directory, flight arrival/departure, and credit card account information, are allowing speech for input and output to enable customers to access this information. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. Systems analysts and programmers design and implement a special interface for each known class of naive users. Usually a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request.

Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

A DBMS is a complex software system. In this section we discuss the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.

2.4.1 DBMS Component Modules

Figure 2.3 illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internal modules of the DBMS responsible for storage of data and processing of transactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system** (**OS**), which schedules disk read/write. Many DBMSs have their own **buffer management** module to schedule disk read/write, because management of buffer storage has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

Let us consider the top part of Figure 2.3 first. It shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands.

The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints.
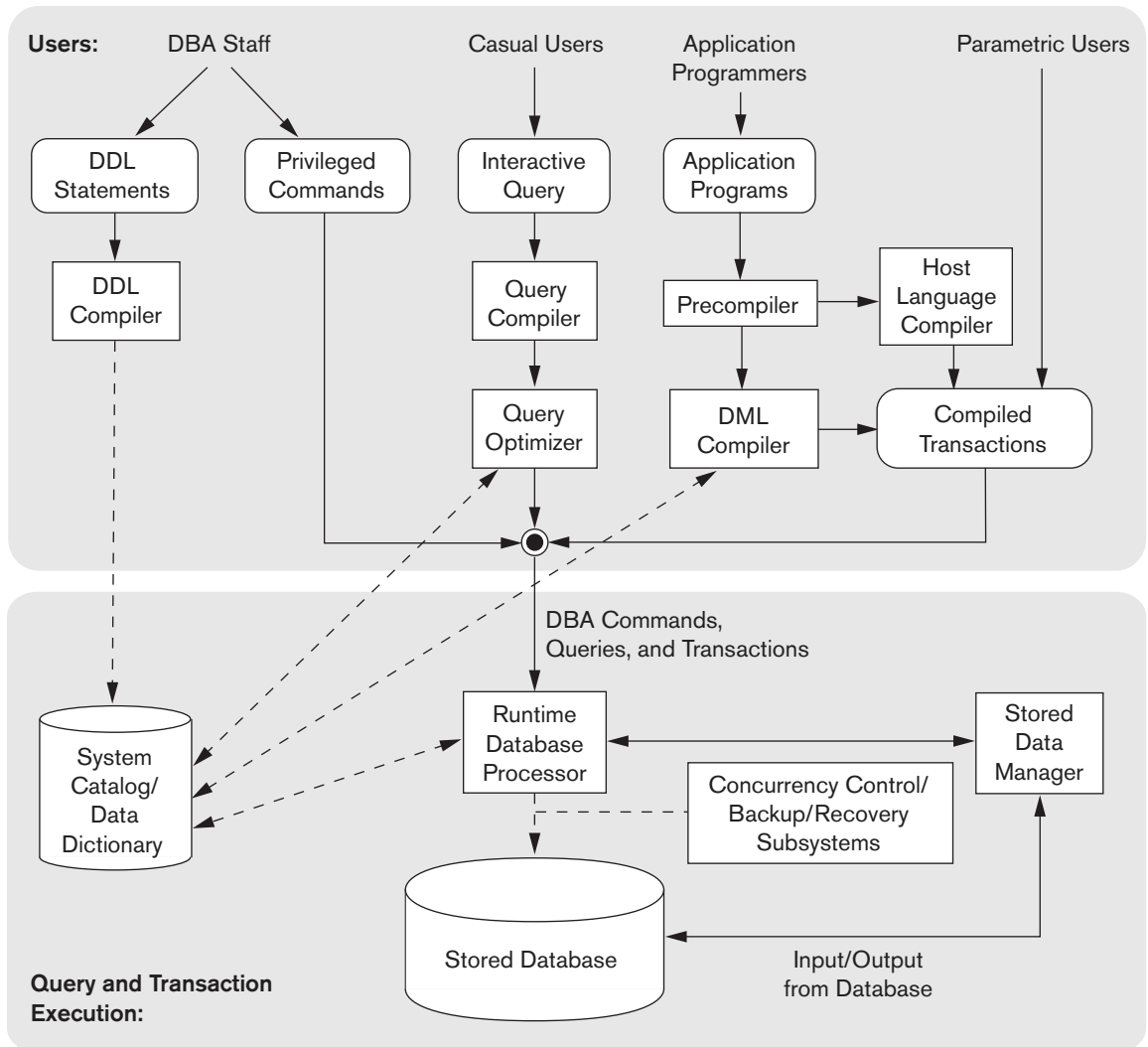
Figure 2.3
Component modules of a DBMS and their interactions.

In addition, the catalog stores many other types of information that are needed by the DBMS modules, which can then look up the catalog information as needed.

Casual users and persons with occasional need for information from the database interact using the **interactive query** interface in Figure 2.3. We have *not explicitly shown* any menu-based or form-based or mobile interactions that are typically used to generate the interactive query automatically or to access canned transactions. These queries are parsed and validated for correctness of the query syntax, the names of files and data elements, and so on by a **query compiler** that compiles

them into an internal form. This internal query is subjected to query optimization (discussed in Chapters 18 and 19). Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of efficient search algorithms during execution. It consults the system catalog for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor. It is also becoming increasingly common to use scripting languages such as PHP and Python to write database programs. Canned transactions are executed repeatedly by parametric users via PCs or mobile apps; these users simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank payment transaction where the account number, payee, and amount may be supplied as parameters.

In the lower part of Figure 2.3, the **runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the **system catalog** and may update it with statistics. It also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module whereas others depend on the OS for buffer management. We have shown **concurrency control** and **backup and recovery systems** separately as a module in this figure. They are integrated into the working of the runtime database processor for purposes of transaction management.

It is common to have the **client program** that accesses the DBMS running on a separate computer or device from the computer on which the database resides. The former is called the **client computer** running DBMS client software and the latter is called the **database server**. In many cases, the client accesses a middle computer, called the **application server**, which in turn accesses the database server. We elaborate on this topic in Section 2.5.

Figure 2.3 is not meant to describe a specific DBMS; rather, it illustrates typical DBMS modules. The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. On the other hand, if the computer system is mainly dedicated to running the database server, the DBMS will control main memory

buffering of disk pages. The DBMS also interfaces with compilers for general-purpose host programming languages, and with application servers and client programs running on separate machines through the system network interface.

2.4.2 Database System Utilities

In addition to possessing the software modules just described, most DBMSs have **database utilities** that help the DBA manage the database system. Common utilities have the following types of functions:

- **Loading.** A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors offer **conversion tools** that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas).

- **Backup.** A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex, but saves storage space.

- **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations and create new access paths to improve performance.

- **Performance monitoring.** Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and the DBMS. CASE tools[12] are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**)

---

[12]Although CASE stands for computer-aided software engineering, many CASE tools are used primarily for database design.

system. In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository**. This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

Application development environments, such as PowerBuilder (Sybase) or JBuilder (Borland), have been quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

The DBMS also needs to interface with **communications software**, whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or personal computers. These are connected to the database site through data communications hardware such as Internet routers, phone lines, long-haul networks, local networks, or satellite communication devices. Many commercial database systems have communication packages that work with the DBMS. The integrated DBMS and data communications system is called a **DB/DC** system. In addition, some distributed DBMSs are physically distributed over multiple machines. In this case, communications networks are needed to connect the machines. These are often **local area networks (LANs)**, but they can also be other types of networks.

2.5 Centralized and Client/Server Architectures for DBMSs

2.5.1 Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Older architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that in older systems, most users accessed the DBMS via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system housing the DBMS, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations, and more recently with mobile devices. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality,
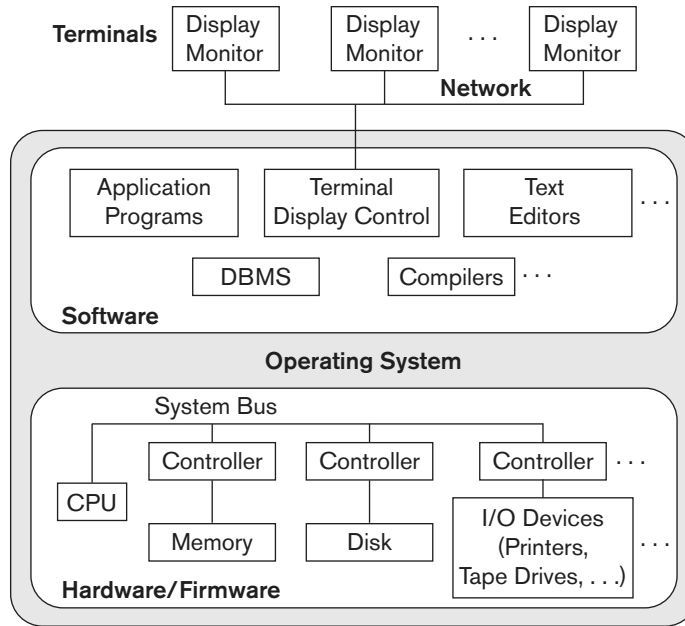
Figure 2.4
A physical centralized architecture.

application program execution, and user interface processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

2.5.2 Basic Client/Server Architectures

First, we discuss client/server architecture in general; then we discuss how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates

Figure 2.5
Logical two-tier
client/server
architecture.
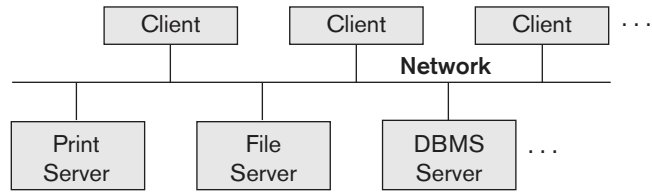
client/server architecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, mobile devices or workstations/PCs that have only client software installed). Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs/workstations and mobile devices as well as a smaller number of server machines, connected via wireless networks or LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at the client, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate

Figure 2.6
Physical two-tier
client/server
architecture.
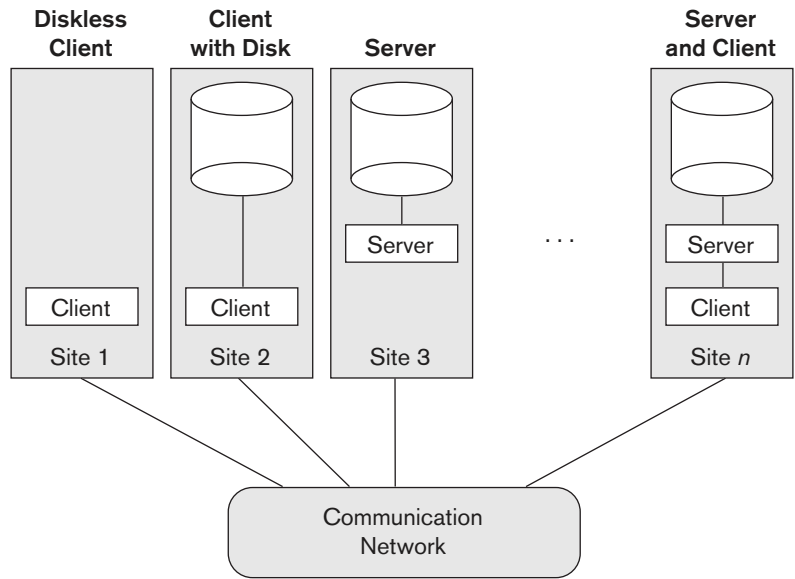
machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: **two-tier** and **three-tier**.[13] We discuss them next.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 6 and 7) provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity** (**ODBC**) provides an **application programming interface** (**API**), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The architectures described here are called **two-tier architectures** because the software components are distributed over two systems: client and server. The advantages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

2.5.4 Three-Tier and *n*-Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7(a).

---

[13]There are many other variations of client/server architectures. We discuss the two most basic ones here.
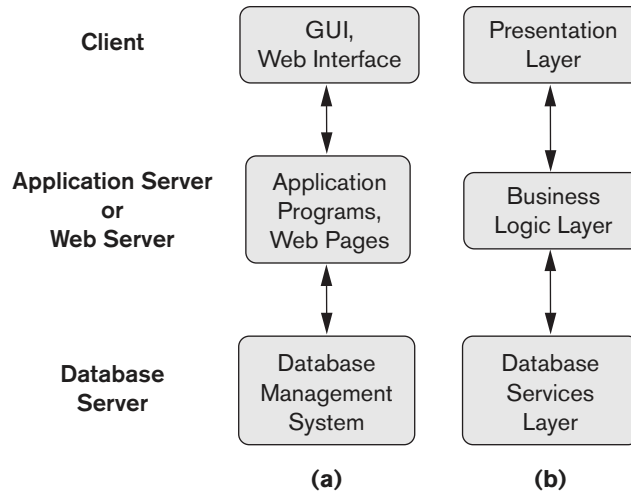
This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by running application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain user interfaces and Web browsers. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to the users. Thus, the *user interface, application rules,* and *data access* act as the three tiers. Figure 2.7(b) shows another view of the three-tier architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side. The client machine is typically a PC or mobile device connected to the Web.

Other architectures have also been proposed. It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to *n*-tier architectures, where *n* may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, *n*-tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can be handled independently. Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a *middleware layer,* which

accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

Advances in encryption and decryption technology make it safer to transfer sensitive data from server to client in encrypted form, where it will be decrypted. The latter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major concern. Various technologies for data compression also help to transfer large amounts of data from servers to clients over wired and wireless networks.

2.6 Classification of Database Management Systems

Several criteria can be used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**, and the systems based on this model are known as **SQL systems.** The **object data model** has been implemented in some commercial systems but has not had widespread use. Recently, so-called **big data systems,** also known as **key-value storage systems** and **NOSQL systems,** use various data models: **document-based, graph-based, column-based,** and **key-value data models.** Many legacy applications still run on database systems based on the **hierarchical** and **network data models**.

The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called **object-relational DBMS**s. We can categorize DBMSs based on the data model: relational, object, object-relational, NOSQL, key-value, hierarchical, network, and other.

Some experimental DBMSs are based on the XML (eXtended Markup Language) model, which is a **tree-structured data model.** These have been called **native XML DBMSs.** Several commercial relational DBMSs have added XML interfaces and storage to their products.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with PCs. **Multiuser systems**, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database reside totally at a single computer site. A **distributed** DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites connected by a computer network. Big data systems are often massively distributed, with hundreds of sites. The data is often replicated on multiple sites so that failure of a site will not make some data unavailable.

Homogeneous DDBMSs use the same DBMS software at all the sites, whereas **heterogeneous** DDBMSs can use different DBMS software at each site. It is also possible to develop **middleware software** to access several autonomous preexisting databases stored under heterogeneous DBMSs. This leads to a **federated** DBMS (or **multidatabase system**), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use client-server architecture, as we described in Section 2.5.

The fourth criterion is cost. It is difficult to propose a classification of DBMSs based on cost. Today we have open source (free) DBMS products like MySQL and PostgreSQL that are supported by third-party vendors with additional services. The main RDBMS products are available as free examination 30-day copy versions as well as personal versions, which may cost under $100 and allow a fair amount of functionality. The giant systems are being sold in modular form with components to handle distribution, replication, parallel processing, mobile capability, and so on, and with a large number of parameters that must be defined for the configuration. Furthermore, they are sold in the form of licenses—site licenses allow unlimited use of the database system with any number of copies running at the customer site. Another type of license limits the number of concurrent users or the number of user seats at a location. Standalone single-user versions of some systems like Microsoft Access are sold per copy or included in the overall configuration of a desktop or laptop. In addition, data warehousing and mining features, as well as support for additional data types, are made available at extra cost. It is possible to pay millions of dollars for the installation and maintenance of large database systems annually.

We can also classify a DBMS on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be **general purpose** or **special purpose**. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of **online transaction processing** (**OLTP**) systems, which must support a large number of concurrent transactions without imposing excessive delays.

Let us briefly elaborate on the main criterion for classifying DBMSs: the data model. The **relational data model** represents a database as a collection of tables, where each table can be stored as a separate file. The database in Figure 1.2 resembles a basic relational representation. Most relational databases use the high-level query language called SQL and support a limited form of user views. We discuss the relational model and its languages and operations in Chapters 5 through 8, and techniques for programming relational applications in Chapters 10 and 11.

The **object data model** defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class**, and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of

each class are specified in terms of predefined procedures called **methods**. Relational DBMSs have been extending their models to incorporate object database concepts and other capabilities; these systems are referred to as **object-relational** or **extended relational systems**. We discuss object databases and object-relational systems in Chapter 12.

Big data systems are based on various data models, with the following four data models most common. The **key-value data model** associates a unique key with each value (which can be a record or object) and provides very fast access to a value given its key. The **document data model** is based on JSON (Java Script Object Notation) and stores the data as documents, which somewhat resemble complex objects. The **graph data model** stores objects as graph nodes and relationships among objects as directed graph edges. Finally, the **column-based data models** store the columns of rows clustered on disk pages for fast access and allow multiple versions of the data. We will discuss some of these in more detail in Chapter 24.

The **XML model** has emerged as a standard for exchanging data over the Web and has been used as a basis for implementing several prototype native XML systems. XML uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements; with the use of tags, data can be nested to create complex tree structures. This model conceptually resembles the object model but uses different terminology. XML capabilities have been added to many commercial DBMS products. We present an overview of XML in Chapter 13.

Two older, historically important data models, now known as **legacy data models**, are the network and hierarchical models. The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type**. A 1:N, or one-to-many, relationship relates one instance of a record to many record instances using some pointer linking mechanism in these models. The network model, also known as the CODASYL DBTG model,[14] has an associated record-at-a-time language that must be embedded in a host programming language. The network DML was proposed in the 1971 Database Task Group (DBTG) Report as an extension of the COBOL language.

The **hierarchical model** represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model. A popular hierarchical DML is DL/1 of the IMS system. It dominated the DBMS market for over 20 years between 1965 and 1985. Its DML, called DL/1, was a de facto industry standard for a long time.[15]

---

[14]CODASYL DBTG stands for Conference on Data Systems Languages Database Task Group, which is the committee that specified the network model and its language.

[15]The full chapters on the network and hierarchical models from the second edition of this book are available from this book's Companion Web site at http://www.aw.com/elmasri.

2.7 Summary

In this chapter we introduced the main concepts used in database systems. We defined a data model and we distinguished three main categories:

- High-level or conceptual data models (based on entities and relationships)
- Low-level or physical data models
- Representational or implementation data models (record-based, object-oriented)

We distinguished the schema, or description of a database, from the database itself. The schema does not change very often, whereas the database state changes every time data is inserted, deleted, or modified. Then we described the three-schema DBMS architecture, which allows three schema levels:

- An internal schema describes the physical storage structure of the database.
- A conceptual schema is a high-level description of the whole database.
- External schemas describe the views of different user groups.

A DBMS that cleanly separates the three levels must have mappings among the schemas to transform requests and query results from one level to the next. Most DBMSs do not separate the three levels completely. We used the three-schema architecture to define the concepts of logical and physical data independence.

Then we discussed the main types of languages and interfaces that DBMSs support. A data definition language (DDL) is used to define the database conceptual schema. In most DBMSs, the DDL also defines user views and, sometimes, storage structures; in other DBMSs, separate languages or functions exist for specifying storage structures. This distinction is fading away in today's relational implementations, with SQL serving as a catchall language to perform multiple roles, including view definition. The storage definition part (SDL) was included in SQL's early versions, but is now typically implemented as special commands for the DBA in relational DBMSs. The DBMS compiles all schema definitions and stores their descriptions in the DBMS catalog.

A data manipulation language (DML) is used for specifying database retrievals and updates. DMLs can be high level (set-oriented, nonprocedural) or low level (record-oriented, procedural). A high-level DML can be embedded in a host programming language, or it can be used as a standalone language; in the latter case it is often called a query language.

We discussed different types of interfaces provided by DBMSs and the types of DBMS users with which each interface is associated. Then we discussed the database system environment, typical DBMS software modules, and DBMS utilities for helping users and the DBA staff perform their tasks. We continued with an overview of the two-tier and three-tier architectures for database applications.

Finally, we classified DBMSs according to several criteria: data model, number of users, number of sites, types of access paths, and cost. We discussed the availability of DBMSs and additional modules—from no cost in the form of open source software to configurations that annually cost millions to maintain. We also pointed out the variety of licensing arrangements for DBMS and related products. The main classification of DBMSs is based on the data model. We briefly discussed the main data models used in current commercial DBMSs.

Review Questions

2.1. Define the following terms: *data model, database schema, database state, internal schema, conceptual schema, external schema, data independence, DDL, DML, SDL, VDL, query language, host language, data sublanguage, database utility, catalog, client/server architecture, three-tier architecture,* and n-*tier architecture.*

2.2. Discuss the main categories of data models. What are the basic differences among the relational model, the object model, and the XML model?

2.3. What is the difference between a database schema and a database state?

2.4. Describe the three-schema architecture. Why do we need mappings among schema levels? How do different schema definition languages support this architecture?

2.5. What is the difference between logical data independence and physical data independence? Which one is harder to achieve? Why?

2.6. What is the difference between procedural and nonprocedural DMLs?

2.7. Discuss the different types of user-friendly interfaces and the types of users who typically use each.

2.8. With what other computer system software does a DBMS interact?

2.9. What is the difference between the two-tier and three-tier client/server architectures?

2.10. Discuss some types of database utilities and tools and their functions.

2.11. What is the additional functionality incorporated in n-tier architecture ($n > 3$)?

Exercises

2.12. Think of different users for the database shown in Figure 1.2. What types of applications would each user need? To which user category would each belong, and what type of interface would each need?

2.13. Choose a database application with which you are familiar. Design a schema and show a sample database for that application, using the notation of Figures 1.2 and 2.1. What types of additional information and constraints would you like to represent in the schema? Think of several users of your database, and design a view for each.

2.14. If you were designing a Web-based system to make airline reservations and sell airline tickets, which DBMS architecture would you choose from Section 2.5? Why? Why would the other architectures not be a good choice?

2.15. Consider Figure 2.1. In addition to constraints relating the values of columns in one table to columns in another table, there are also constraints that impose restrictions on values in a column or a combination of columns within a table. One such constraint dictates that a column or a group of columns must be unique across all rows in the table. For example, in the STUDENT table, the Student_number column must be unique (to prevent two different students from having the same Student_number). Identify the column or the group of columns in the other tables that must be unique across all rows in the table.

Selected Bibliography

Many database textbooks, including Date (2004), Silberschatz et al. (2011), Ramakrishnan and Gehrke (2003), Garcia-Molina et al. (2002, 2009), and Abiteboul et al. (1995), provide a discussion of the various database concepts presented here. Tsichritzis and Lochovsky (1982) is an early textbook on data models. Tsichritzis and Klug (1978) and Jardine (1977) present the three-schema architecture, which was first suggested in the DBTG CODASYL report (1971) and later in an American National Standards Institute (ANSI) report (1975). An in-depth analysis of the relational data model and some of its possible extensions is given in Codd (1990). The proposed standard for object-oriented databases is described in Cattell et al. (2000). Many documents describing XML are available on the Web, such as XML (2005).

Examples of database utilities are the ETI Connect, Analyze and Transform tools (http://www.eti.com) and the database administration tool, DBArtisan, from Embarcadero Technologies (http://www.embarcadero.com).

Conceptual Data Modeling and Database Design

This page intentionally left blank

Data Modeling Using the Entity–Relationship (ER) Model

Conceptual modeling is a very important phase in designing a successful database application. Generally, the term **database application** refers to a particular database and the associated programs that implement the database queries and updates. For example, a BANK database application that keeps track of customer accounts would include programs that implement database updates corresponding to customer deposits and withdrawals. These programs would provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application—the bank customers or bank tellers in this example. In addition, it is now common to provide interfaces to these programs to BANK customers via mobile devices using **mobile apps**. Hence, a major part of the database application will require the design, implementation, and testing of these application programs. Traditionally, the design and testing of **application programs** has been considered to be part of *software engineering* rather than *database design*. In many software design tools, the database design methodologies and software engineering methodologies are intertwined since these activities are strongly related.

In this chapter, we follow the traditional approach of concentrating on the database structures and constraints during conceptual database design. The design of application programs is typically covered in software engineering courses. We present the modeling concepts of the **entity–relationship** (**ER**) **model**, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications. We also present the diagrammatic notation associated with the ER model, known as **ER diagrams**.

Object modeling methodologies such as the **Unified Modeling Language** (**UML**) are becoming increasingly popular in both database and software design. These methodologies go beyond database design to specify detailed design of software modules and their interactions using various types of diagrams. An important part of these methodologies—namely, *class diagrams*[1]—is similar in many ways to the ER diagrams. In class diagrams, *operations* on objects are specified, in addition to specifying the database schema structure. Operations can be used to specify the *functional requirements* during database design, as we will discuss in Section 3.1. We present some of the UML notation and concepts for class diagrams that are particularly relevant to database design in Section 3.8, and we briefly compare these to ER notation and concepts. Additional UML notation and concepts are presented in Section 4.6.

This chapter is organized as follows: Section 3.1 discusses the role of high-level conceptual data models in database design. We introduce the requirements for a sample database application in Section 3.2 to illustrate the use of concepts from the ER model. This sample database is used throughout the text. In Section 3.3 we present the concepts of entities and attributes, and we gradually introduce the diagrammatic technique for displaying an ER schema. In Section 3.4 we introduce the concepts of binary relationships and their roles and structural constraints. Section 3.5 introduces weak entity types. Section 3.6 shows how a schema design is refined to include relationships. Section 3.7 reviews the notation for ER diagrams, summarizes the issues and common pitfalls that occur in schema design, and discusses how to choose the names for database schema constructs such as entity types and relationship types. Section 3.8 introduces some UML class diagram concepts, compares them to ER model concepts, and applies them to the same COMPANY database example. Section 3.9 discusses more complex types of relationships. Section 3.10 summarizes the chapter.

The material in Sections 3.8 and 3.9 may be excluded from an introductory course. If a more thorough coverage of data modeling concepts and conceptual database design is desired, the reader should continue to Chapter 4, where we describe extensions to the ER model that lead to the enhanced–ER (EER) model, which includes concepts such as specialization, generalization, inheritance, and union types (categories).

3.1 Using High-Level Conceptual Data Models for Database Design

Figure 3.1 shows a simplified overview of the database design process. The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify

---

[1]A **class** is similar to an *entity type* in many ways.
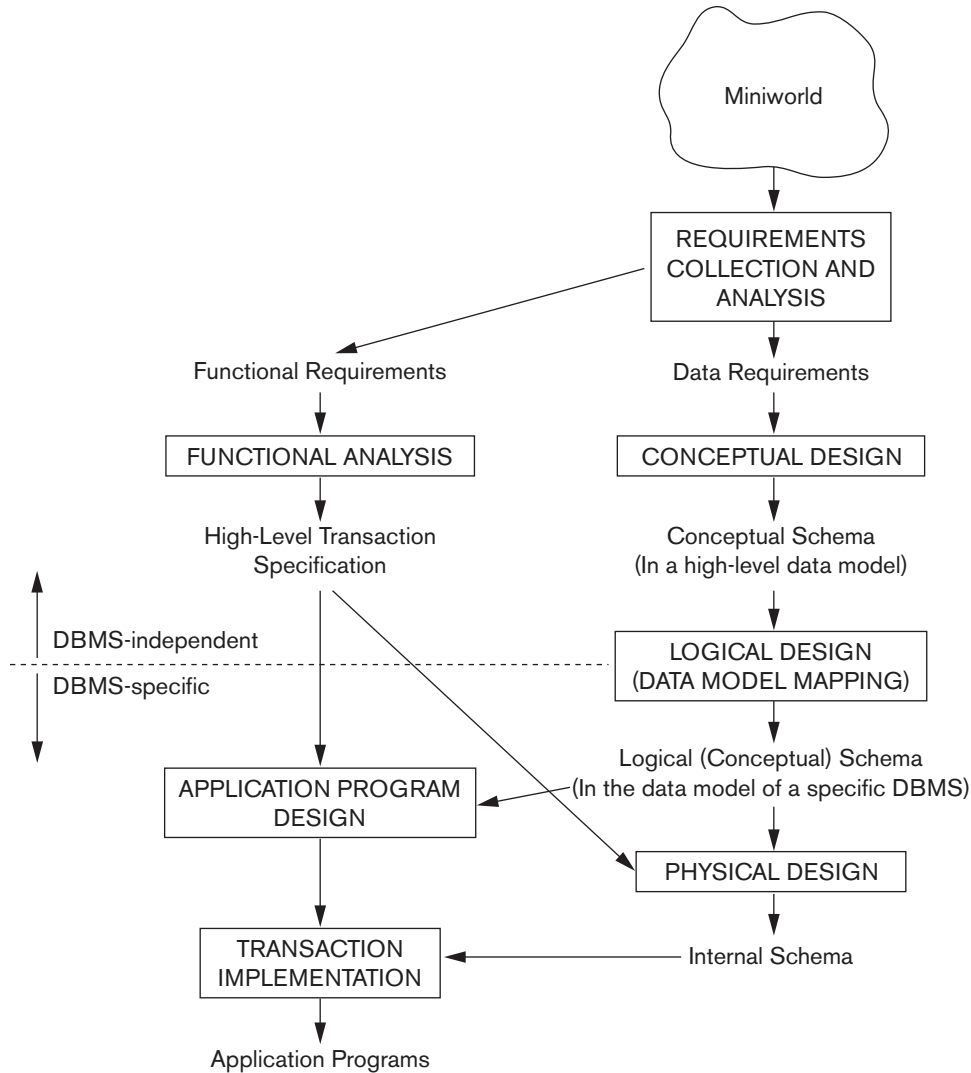
Figure 3.1
A simplified diagram to illustrate the main phases of database design.

the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques to specify functional requirements. We will not discuss any of these techniques here; they are usually described in detail in software engineering texts.

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This

step is called **conceptual design**. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details, which makes it is easier to create a good conceptual database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational (SQL) model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**; its result is a database schema in the implementation data model of the DBMS. Data model mapping is often automated or semiautomated within the database design tools.

The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications.

We present only the basic ER model concepts for conceptual schema design in this chapter. Additional modeling concepts are discussed in Chapter 4, when we introduce the EER model.

3.2 A Sample Database Application

In this section we describe a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design. We list the data requirements for the database here, and then create its conceptual schema step-by-step as we introduce the modeling concepts of the ER model. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the *miniworld*—the part of the company that will be represented in the database.

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.

- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.

- The database will store each employee's name, Social Security number,[2] address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. It is required to keep track of the current number of hours per week that an employee works on each project, as well as the direct supervisor of each employee (who is another employee).

- The database will keep track of the dependents of each employee for insurance purposes, including each dependent's first name, sex, birth date, and relationship to the employee.

Figure 3.2 shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams**. This figure will be explained gradually as the ER model concepts are presented. We describe the step-by-step process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts.

3.3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as *entities*, *relationships*, and *attributes*. In Section 3.3.1 we introduce the concepts of entities and their attributes. We discuss entity types and key attributes in Section 3.3.2. Then, in Section 3.3.3, we specify the initial conceptual design of the entity types for the COMPANY database. We describe relationships in Section 3.4.

3.3.1 Entities and Attributes

Entities and Their Attributes.   The basic concept that the ER model represents is an **entity**, which is a *thing* or *object* in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course). Each entity has **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job. A particular entity

---

[2]The Social Security number, or SSN, is a unique nine-digit identifier assigned to each individual in the United States to keep track of his or her employment, benefits, and taxes. Other countries may have similar identification schemes, such as personal identification card numbers.
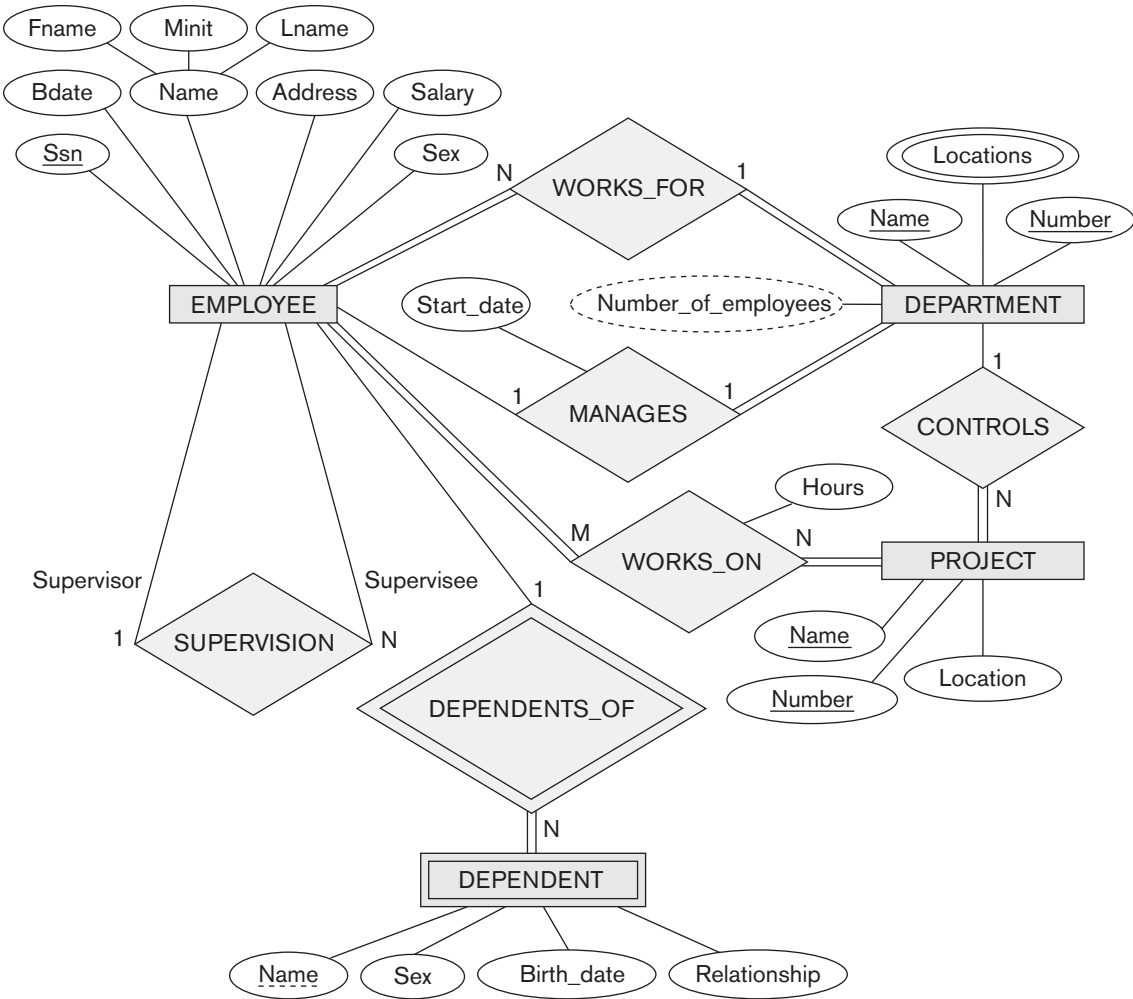
Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 3.14.

will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Figure 3.3 shows two entities and the values of their attributes. The EMPLOYEE entity e_1 has four attributes: Name, Address, Age, and Home_phone; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001', '55', and '713-749-2630', respectively. The COMPANY entity c_1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively.
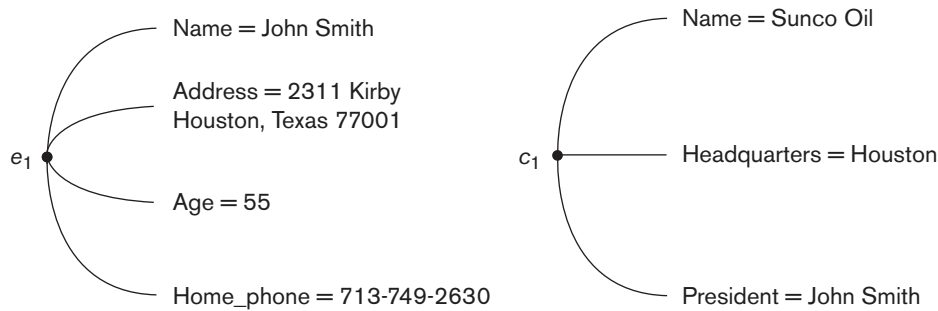
Figure 3.3
Two entities, EMPLOYEE e_1, and COMPANY c_1, and their attributes.

Several types of attributes occur in the ER model: *simple* versus *composite*, *single-valued* versus *multivalued*, and *stored* versus *derived*. First we define these attribute types and illustrate their use via examples. Then we discuss the concept of a *NULL value* for an attribute.

Composite versus Simple (Atomic) Attributes. **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure 3.3 can be subdivided into Street_address, City, State, and Zip,[3] with the values '2311 Kirby', 'Houston', 'Texas', and '77001'. Attributes that are not divisible are called **simple** or **atomic attributes**. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number, as shown in Figure 3.4. The value of a composite attribute is the concatenation of the values of its component simple attributes.

Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its
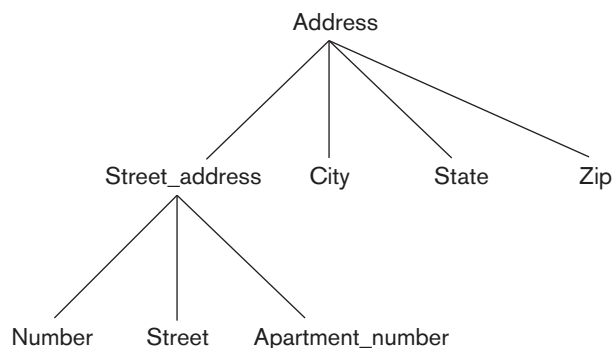
Figure 3.4
A hierarchy of composite attributes.

---

[3]Zip Code is the name used in the United States for a five-digit postal code, such as 76019, which can be extended to nine digits, such as 76019-0015. We use the five-digit Zip in our examples.

components. If the composite attribute is referenced only as a whole, there is no need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (Zip Code, street, and so on), then the whole address can be designated as a simple attribute.

Single-Valued versus Multivalued Attributes. Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values. Similarly, one person may not have any college degrees, another person may have one, and a third person may have two or more degrees; therefore, different people can have different *numbers* of *values* for the College_degrees attribute. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the *number of values* allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and two values, if we assume that a car can have two colors at most.

Stored versus Derived Attributes. In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the Birth_date attribute, which is called a **stored attribute**. Some attribute values can be derived from *related entities*; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that department.

NULL Values. In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called NULL is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. NULL can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone number of 'John Smith' in Figure 3.3. The meaning of the former type of NULL is *not applicable*, whereas the meaning of the latter is *unknown*. The *unknown* category of NULL can be further classified into two cases. The first case arises when it is known that the attribute value exists but is *missing*—for instance, if the Height attribute of a person is listed as NULL. The second case arises when it is *not known* whether the attribute value exists—for example, if the Home_phone attribute of a person is NULL.

Complex Attributes. Notice that, in general, composite and multivalued attributes can be nested arbitrarily. We can represent arbitrary nesting by grouping

{Address_phone({Phone(Area_code,Phone_number)},Address(Street_address (Number,Street,Apartment_number),City,State,Zip))}

Figure 3.5
A complex attribute: Address_phone.

components of a composite attribute between parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces { }. Such attributes are called **complex attributes**. For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified as shown in Figure 3.5.[4] Both Phone and Address are themselves composite attributes.

3.3.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets. A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its *own value(s)* for each attribute. An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. Figure 3.6 shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for each. A few individual entities of each type are also illustrated, along with the values of their attributes. The collection of all entities of a particular entity type in the

Entity Type Name:	EMPLOYEE	COMPANY
	Name, Age, Salary	Name, Headquarters, President

	e_1 •	c_1 •
	(John Smith, 55, 80k)	(Sunco Oil, Houston, John Smith)
Entity Set: (Extension)	e_2 •	c_2 •
	(Fred Brown, 40, 30K)	(Fast Computer, Dallas, Bob King)
	e_3 •	⋮
	(Judy Clark, 25, 20K)	
	⋮	

Figure 3.6
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

---
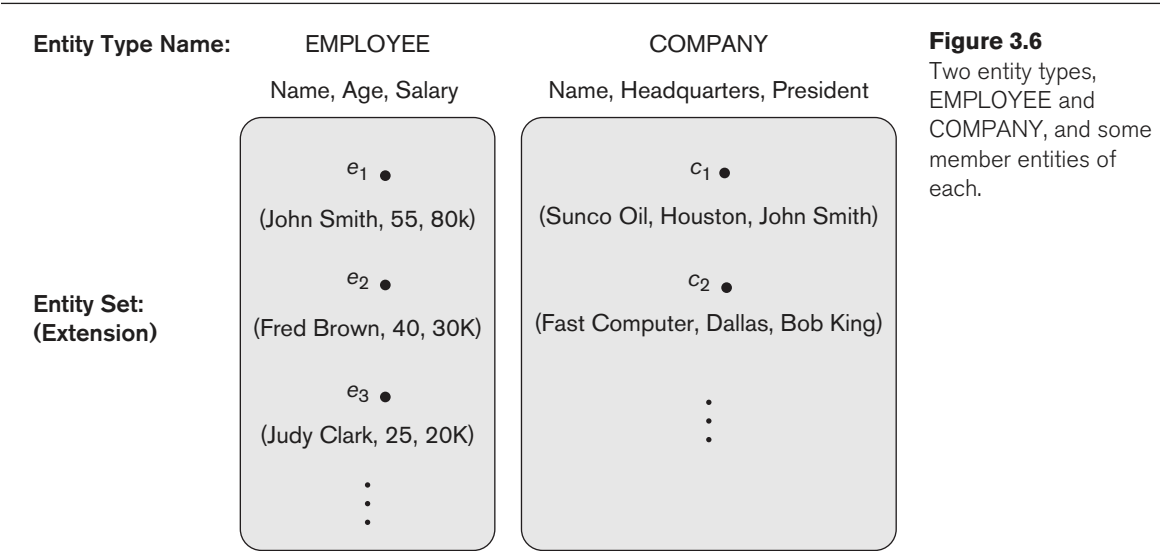
[4]For those familiar with XML, we should note that complex attributes are similar to complex elements in XML (see Chapter 13).

database at any point in time is called an **entity set** or **entity collection**; the entity set is usually referred to using the same name as the entity type, even though they are two separate concepts. For example, EMPLOYEE refers to both a *type of entity* as well as the current collection *of all employee entities* in the database. It is now more common to give separate names to the entity type and entity collection; for example in object and object-relational data models (see Chapter 12).

An entity type is represented in ER diagrams[5] (see Figure 3.2) as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals. Figure 3.7(a) shows a CAR entity type in this notation.

An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure. The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 3.6 because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is Ssn (Social Security number). Sometimes several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a *composite attribute* and designate it as a key attribute of the entity type. Notice that such a composite key must be *minimal*; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure 3.7(a).

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every entity set* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular entity set; rather, it is a constraint on *any entity set* of the entity type at any point in time. This key constraint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.

Some entity types have *more than one* key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 3.7) is a key in

---

[5]We use a notation for ER diagrams that is close to the original proposed notation (Chen, 1976). Many other notations are in use; we illustrate some of them later in this chapter when we present UML class diagrams, and some additional diagrammatic notations are given in Appendix A.

(a)

(b)
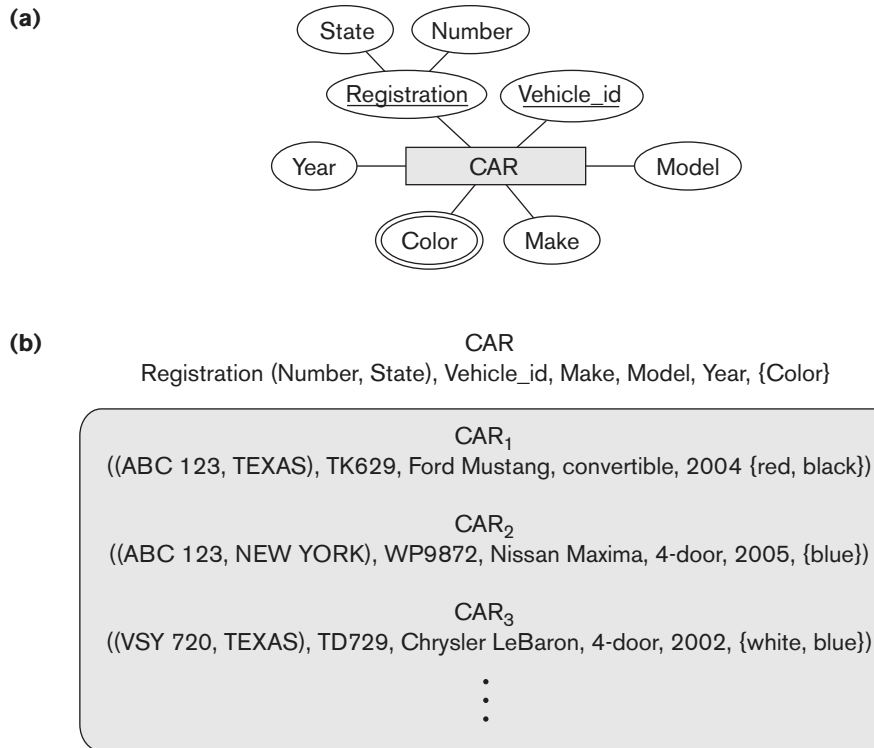
Figure 3.7
The CAR entity type
with two key attributes,
Registration and
Vehicle_id. (a) ER
diagram notation.
(b) Entity set with
three entities.

its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own. An entity type may also have *no key*, in which case it is called a *weak entity type* (see Section 3.5).

In our diagrammatic notation, if two attributes are underlined separately, then *each is a key on its own*. Unlike the relational model (see Section 5.2.2), there is no concept of primary key in the ER model that we present here; the primary key will be chosen during mapping to a relational schema (see Chapter 9).

Value Sets (Domains) of Attributes. Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity. In Figure 3.6, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute to be the set of strings of alphabetic characters separated by blank characters, and so on. Value sets are not typically displayed in basic ER diagrams and are similar to the basic **data types** available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on. However, data types of attributes can

be specified in UML class diagrams (see Section 3.8) and in other diagrammatic notations used in database design tools. Additional data types to represent common database types, such as date, time, and other concepts, are also employed.

Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set[6] $P(V)$ of V:

$$A : E \rightarrow P(V)$$

We refer to the value of attribute A for entity e as $A(e)$. The previous definition covers both single-valued and multivalued attributes, as well as NULLs. A NULL value is represented by the *empty set*. For single-valued attributes, $A(e)$ is restricted to being a *singleton set* for each entity e in E, whereas there is no restriction on multivalued attributes.[7] For a composite attribute A, the value set V is the power set of the Cartesian product of $P(V_1)$, $P(V_2)$, . . . , $P(V_n)$, where V_1, V_2, . . . , V_n are the value sets of the simple component attributes that form A:

$$V = P(P(V_1) \times P(V_2) \times \cdots \times P(V_n))$$

The value set provides all possible values. Usually only a small number of these values exist in the database at a particular time. Those values represent the data from the current state of the miniworld and correspond to the data as it actually exists in the miniworld.

3.3.3 Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database, based on the requirements described in Section 3.2. After defining several entity types and their attributes here, we refine our design in Section 3.4 after we introduce the concept of a relationship. According to the requirements listed in Section 3.2, we can identify four entity types—one corresponding to each of the four items in the specification (see Figure 3.8):

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.

2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.

3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First_name, Middle_initial, Last_name—or of Address. In

---

[6]The **power set** $P(V)$ of a set V is the set of all subsets of V.

[7]A **singleton** set is a set with only one element (value).
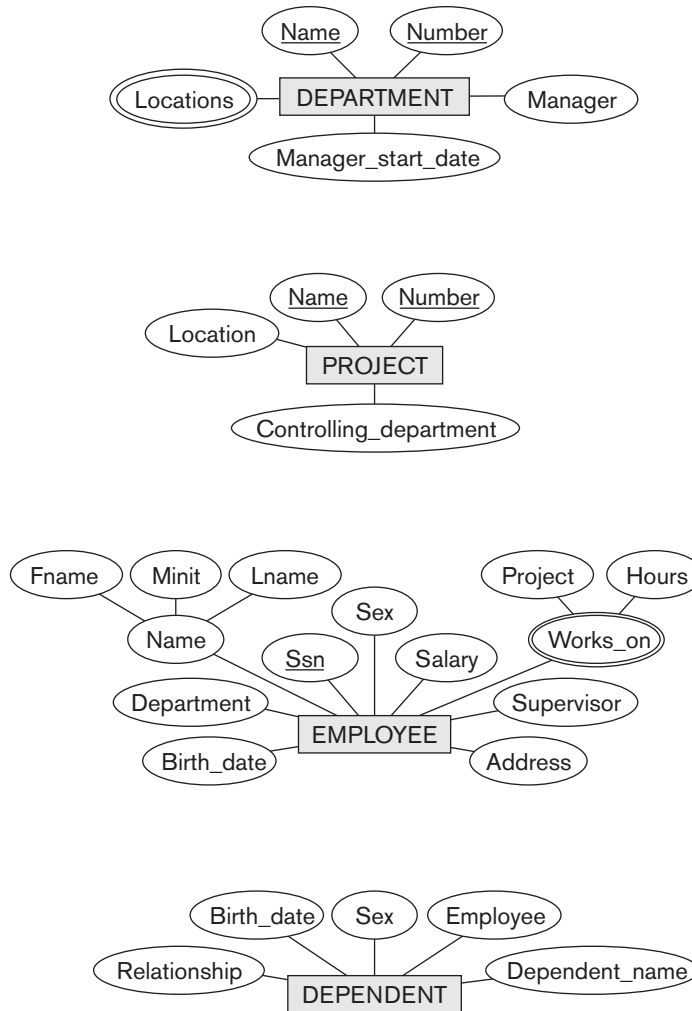
Figure 3.8
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

our example, Name is modeled as a composite attribute, whereas Address is not, presumably after consultation with the users.

4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

Another requirement is that an employee can work on several projects, and the database has to store the number of hours per week an employee works on each project. This requirement is listed as part of the third requirement in Section 3.2, and it can be represented by a multivalued composite attribute of EMPLOYEE called Works_on with the simple components (Project, Hours). Alternatively, it can be represented as a multivalued composite attribute of PROJECT called Workers with the simple components (Employee, Hours). We choose the first

alternative in Figure 3.8; we shall see in the next section that this will be refined into a many-to-many relationship, once we introduce the concepts of relationships.

3.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints

In Figure 3.8 there are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department; the attribute Controlling_department of PROJECT refers to the department that controls the project; the attribute Supervisor of EMPLOYEE refers to another employee (the one who supervises this employee); the attribute Department of EMPLOYEE refers to the department for which the employee works; and so on. In the ER model, these references should not be represented as attributes but as **relationships**. The initial COMPANY database schema from Figure 3.8 will be refined in Section 3.6 to represent relationships explicitly. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

This section is organized as follows: Section 3.4.1 introduces the concepts of relationship types, relationship sets, and relationship instances. We define the concepts of relationship degree, role names, and recursive relationships in Section 3.4.2, and then we discuss structural constraints on relationships—such as cardinality ratios and existence dependencies—in Section 3.4.3. Section 3.4.4 shows how relationship types can also have attributes.

3.4.1 Relationship Types, Sets, and Instances

A **relationship type** R among n entity types $E_1, E_2, \ldots, E_n$ defines a set of associations—or a **relationship set**—among entities from these entity types. Similar to the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name*, R. Mathematically, the relationship set R is a set of **relationship instances** r_i, where each r_i associates n individual entities $(e_1, e_2, \ldots, e_n)$, and each entity e_j in r_i is a member of entity set E_j, $1 \leq j \leq n$. Hence, a relationship set is a mathematical relation on $E_1, E_2, \ldots, E_n$; alternatively, it can be defined as a subset of the Cartesian product of the entity sets $E_1 \times E_2 \times \ldots \times E_n$. Each of the entity types $E_1, E_2, \ldots, E_n$ is said to **participate** in the relationship type R; similarly, each of the individual entities $e_1, e_2, \ldots, e_n$ is said to **participate** in the relationship instance $r_i = (e_1, e_2, \ldots, e_n)$.

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i are related in some way in the corresponding miniworld situation. For example, consider a relationship type WORKS_FOR between the two entity types
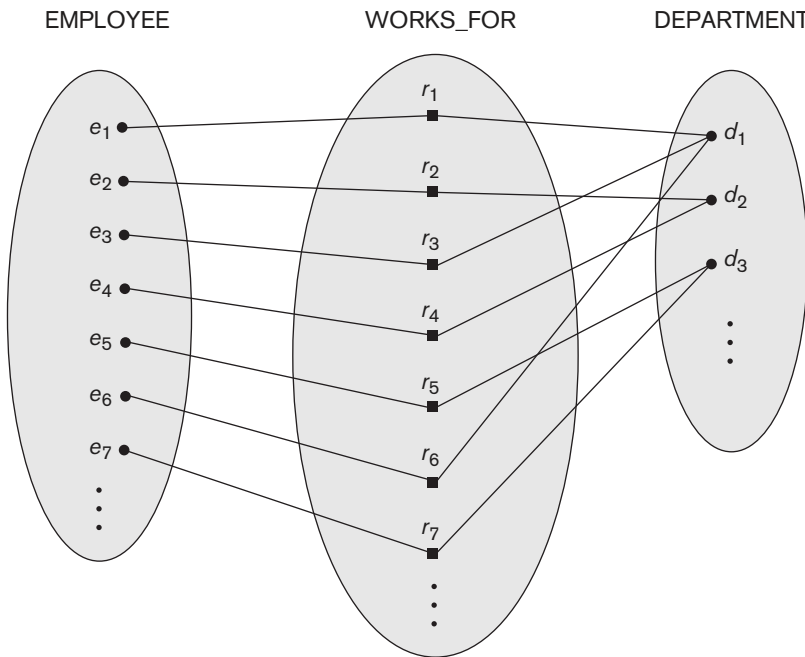
Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity. Figure 3.9 illustrates this example, where each relationship instance r_i is shown connected to the EMPLOYEE and DEPARTMENT entities that participate in r_i. In the miniworld represented by Figure 3.9, the employees e_1, e_3, and e_6 work for department d_1; the employees e_2 and e_4 work for department d_2; and the employees e_5 and e_7 work for department d_3.

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box (see Figure 3.2).

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type. The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary**, and one of degree three is called **ternary**. An example of a ternary relationship is SUPPLY, shown in Figure 3.10, where each relationship instance r_i associates three entities—a supplier s, a part p, and a project j—whenever s supplies part p to project j. Relationships can
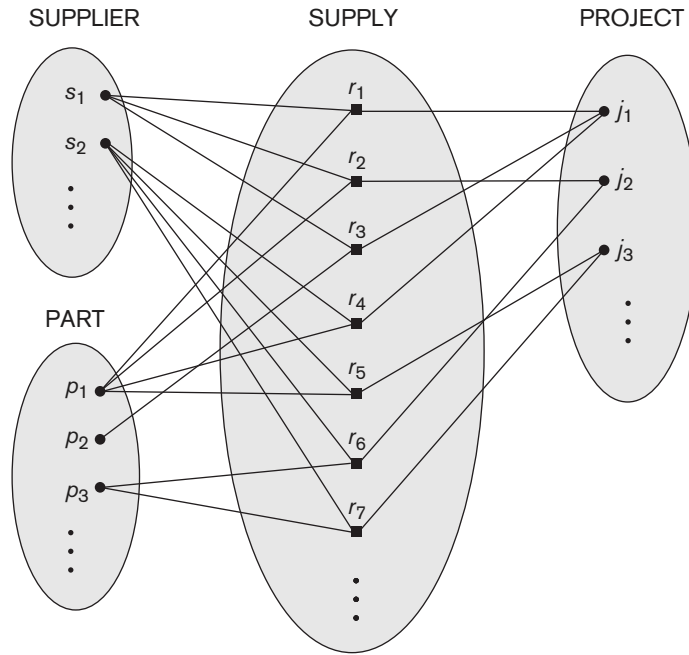
Figure 3.10
Some relationship instances in the SUPPLY ternary relationship set.

generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships; we characterize them further in Section 3.9.

Relationships as Attributes. It is sometimes convenient to think of a binary relationship type in terms of attributes, as we discussed in Section 3.3.3. Consider the WORKS_FOR relationship type in Figure 3.9. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is (a reference to) the DEPARTMENT entity for which that employee works. Hence, the value set for this Department attribute is the set of *all* DEPARTMENT entities, which is the DEPARTMENT entity set. This is what we did in Figure 3.8 when we specified the initial design of the entity type EMPLOYEE for the COMPANY database. However, when we think of a binary relationship as an attribute, we always have two options or two points of view. In this example, the alternative point of view is to think of a multivalued attribute Employees of the entity type DEPARTMENT whose value for each DEPARTMENT entity is the set of EMPLOYEE entities who work for that department. The value set of this Employees attribute is the power set of the EMPLOYEE entity set. Either of these two attributes—Department of EMPLOYEE or Employees of DEPARTMENT—can represent the WORKS_FOR relationship type. If both are represented, they are constrained to be inverses of each other.[8]

---

[8]This concept of representing relationship types as attributes is used in a class of data models called **functional data models**. In object databases (see Chapter 12), relationships can be represented by reference attributes, either in one direction or in both directions as inverses. In relational databases (see Chapter 5), foreign keys are a type of reference attribute used to represent relationships.

Role Names and Recursive Relationships. Each entity type that participates in a relationship type plays a particular role in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and it helps to explain what the relationship means. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each participating entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for distinguishing the meaning of the role that each participating entity plays. Such relationship types are called **recursive relationships** or **self-referencing relationships**. Figure 3.11 shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or *boss*), and once in the role of *supervisee* (or *subordinate*). Each relationship instance r_i in SUPERVISION associates two different employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of supervisee. In Figure 3.11, the lines marked '1' represent the supervisor role, and those marked '2' represent the supervisee role; hence, e_1 supervises e_2 and e_3, e_4 supervises e_6 and e_7, and e_5 supervises e_1 and e_4. In this example, each relationship instance must be connected with two lines, one marked with '1' (supervisor) and the other with '2' (supervisee).
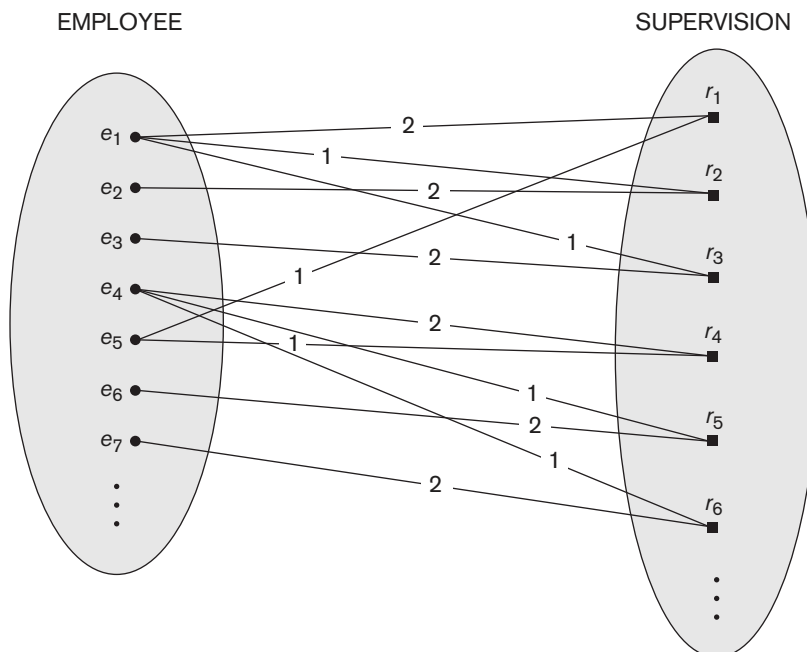
Figure 3.11
A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

3.4.3 Constraints on Binary Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationships represent. For example, in Figure 3.9, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of binary relationship constraints: *cardinality ratio* and *participation*.

Cardinality Ratios for Binary Relationships. The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to (that is, employs) any number of employees (N),[9] but an employee can be related to (work for) at most one department (1). This means that for this particular relationship type WORKS_FOR, a particular department entity can be related to any number of employees (N indicates there is no maximum number). On the other hand, an employee can be related to a maximum of one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES (Figure 3.12), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage at

Figure 3.12
A 1:1 relationship,
MANAGES.

EMPLOYEE   MANAGES   DEPARTMENT

---
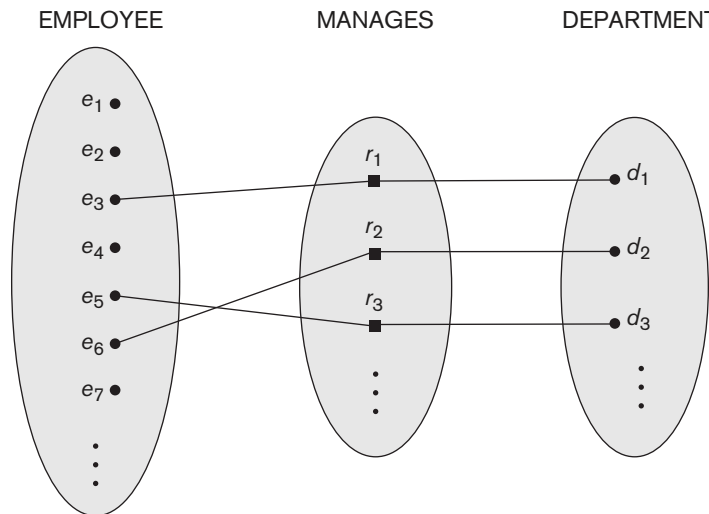
[9]N stands for *any number* of related entities (zero or more). In some notations, the asterisk symbol (*) is used instead of N.

EMPLOYEE   WORKS_ON   PROJECT
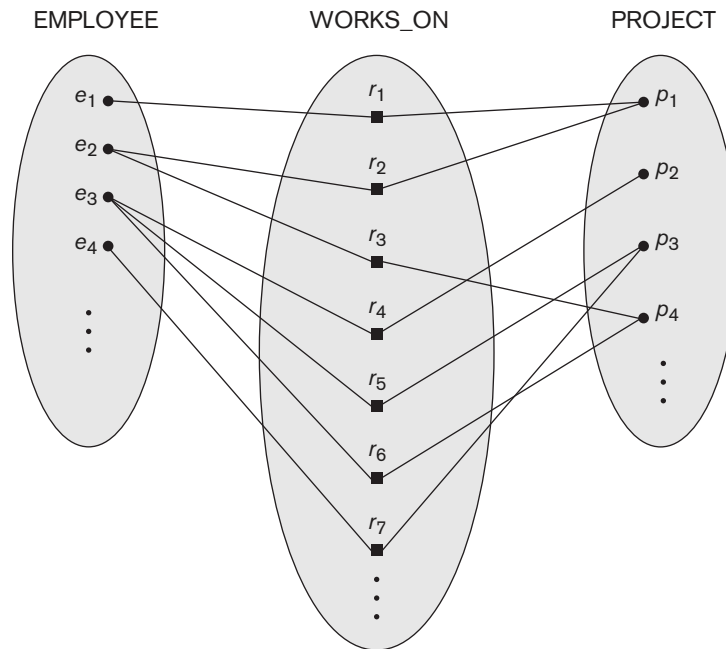
most one department and a department can have at most one manager. The relationship type WORKS_ON (Figure 3.13) is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.

Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 3.2. Notice that in this notation, we can either specify no maximum (N) or a maximum of one (1) on participation. An alternative notation (see Section 3.7.4) allows the designer to specify a specific *maximum number* on participation, such as 4 or 5.

Participation Constraints and Existence Dependencies. The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the *minimum* number of relationship instances that each entity can participate in and is sometimes called the **minimum cardinality constraint**. There are two types of participation constraints—total and partial—that we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 3.9). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation**, meaning that every entity in *the total set* of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency**. In Figure 3.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the

MANAGES relationship type is **partial**, meaning that *some* or *part of the set of* employee entities are related to some department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation (or existence dependency) is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 3.2). Notice that in this notation, we can either specify no minimum (partial participation) or a minimum of one (total participation). An alternative notation (see Section 3.7.4) allows the designer to specify a specific *minimum number* on participation in the relationship, such as 4 or 5.

We will discuss constraints on higher-degree relationships in Section 3.9.

3.4.4 Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that a particular employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type in Figure 3.13. Another example is to include the date on which a manager started managing a department via an attribute Start_date for the MANAGES relationship type in Figure 3.12.

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the Start_date attribute for the MANAGES relationship can be an attribute of either EMPLOYEE (manager) or DEPARTMENT, although conceptually it belongs to MANAGES. This is because MANAGES is a 1:1 relationship, so every department or employee entity participates in *at most one* relationship instance. Hence, the value of the Start_date attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type on the N-side of the relationship. For example, in Figure 3.9, if the WORKS_FOR relationship also has an attribute Start_date that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee works for at most one department, and hence participates in at most one relationship instance in WORKS_FOR, but a department can have many employees, each with a different start date. In both 1:1 and 1:N relationship types, the decision where to place a relationship attribute—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

For M:N (many-to-many) relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes *must be specified as relationship attributes*. An example is the Hours attribute of the M:N relationship WORKS_ON (Figure 3.13); the number of hours per week an employee currently works on a project is determined by an employee-project combination and not separately by either entity.

3.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types**. In contrast, **regular entity types** that do have a key attribute—which include all the examples discussed so far—are called **strong entity types**. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying** or **owner entity type**,[10] and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.[11] A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (License_number) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 3.2). In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, Birth_date, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to *own* the dependent entities that are related to it.

A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are *related to the same owner entity*.[12] In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 3.2). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a multivalued composite attribute with the component attributes Name, Birth_date, Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the weak entity type representation if the weak entity type participates independently in relationship types other than its identifying relationship type.

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we illustrate in Section 3.9.

---

[10]The identifying entity type is also sometimes called the **parent entity type** or the **dominant entity type**.

[11]The weak entity type is also sometimes called the **child entity type** or the **subordinate entity type**.

[12]The partial key is sometimes called the **discriminator**.

3.6 Refining the ER Design for the COMPANY Database

We can now refine the database design in Figure 3.8 by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements listed in Section 3.2. If some cardinality ratio or dependency cannot be determined from the requirements, the users must be questioned further to determine these structural constraints.

In our example, we specify the following relationship types:

- MANAGES, which is a 1:1(one-to-one) relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.[13] The attribute Start_date is assigned to this relationship type.

- WORKS_FOR, a 1:N (one-to-many) relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.

- CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.

- SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.

- WORKS_ON, determined to be an M:N (many-to-many) relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.

- DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

After specifying the previous six relationship types, we remove from the entity types in Figure 3.8 all attributes that have been refined into relationships. These include Manager and Manager_start_date from DEPARTMENT; Controlling_department from PROJECT; Department, Supervisor, and Works_on from EMPLOYEE; and Employee from DEPENDENT. It is important to have the least possible redundancy when we design the conceptual schema of a database. If some redundancy is desired at the storage level or at the user view level, it can be introduced later, as discussed in Section 1.6.1.

---

[13]The rules in the miniworld that determine the constraints are sometimes called the *business rules*, since they are determined by the *business* or organization that will utilize the database.

3.7 ER Diagrams, Naming Conventions, and Design Issues

3.7.1 Summary of Notation for ER Diagrams

Figures 3.9 through 3.13 illustrate examples of the participation of entity types in relationship types by displaying their entity sets and relationship sets (or extensions)—the individual entity instances in an entity set and the individual relationship instances in a relationship set. In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful in database design because a database schema changes rarely, whereas the contents of the entity sets may change frequently. In addition, the schema is obviously easier to display, because it is much smaller.

Figure 3.2 displays the COMPANY **ER database schema** as an **ER diagram**. We now review the full ER diagram notation. Regular (strong) entity types such as EMPLOYEE, DEPARTMENT, and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the Name attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the Number_of_employees attribute of DEPARTMENT.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type is underlined with a dotted line.

In Figure 3.2 the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT:EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for DEPARTMENT: EMPLOYEE in WORKS_FOR, and M:N for WORKS_ON. The participation constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

In Figure 3.2 we show the role names for the SUPERVISION relationship type because the same EMPLOYEE entity type plays two distinct roles in that relationship. Notice that the cardinality ratio is 1:N from supervisor to supervisee because each employee in the role of supervisee has at most one direct supervisor, whereas an employee in the role of supervisor can supervise zero or more employees.

Figure 3.14 summarizes the conventions for ER diagrams. It is important to note that there are many other alternative diagrammatic notations (see Section 3.7.4 and Appendix A).

3.7.2 Proper Naming of Schema Constructs

When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) roles is not always straightforward. One should choose names that convey, as much as possible, the meanings attached to the different constructs in the schema. We choose to use *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type. In our ER diagrams, we will use the convention that entity type and relationship type names are in uppercase letters, attribute names have their initial letter capitalized, and role names are in lowercase letters. We have used this convention in Figure 3.2.

As a general practice, given a narrative description of the database requirements, the *nouns* appearing in the narrative tend to give rise to entity type names, and the *verbs* tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.
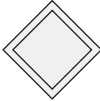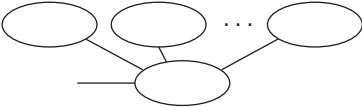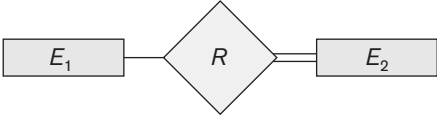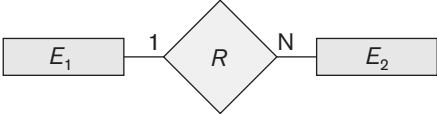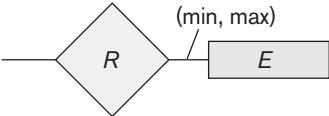
Another naming consideration involves choosing binary relationship names to make the ER diagram of the schema readable from left to right and from top to bottom. We have generally followed this guideline in Figure 3.2. To explain this naming convention further, we have one exception to the convention in Figure 3.2—the DEPENDENTS_OF relationship type, which reads from bottom to top. When we describe this relationship, we can say that the DEPENDENT entities (bottom entity type) are DEPENDENTS_OF (relationship name) an EMPLOYEE (top entity type). To change this to read from top to bottom, we could rename the relationship type to HAS_DEPENDENTS, which would then read as follows: An EMPLOYEE entity (top entity type) HAS_DEPENDENTS (relationship name) of type DEPENDENT (bottom entity type). Notice that this issue arises because each binary relationship can be described starting from either of the two participating entity types, as discussed in the beginning of Section 3.4.

3.7.3 Design Choices for ER Conceptual Design

It is occasionally difficult to decide whether a particular concept in the miniworld should be modeled as an entity type, an attribute, or a relationship type. In this section, we give some brief guidelines as to which construct should be chosen in particular situations.

In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached. Some of the refinements that are often used include the following:

- A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is often the case that a pair of such attributes that are inverses of one another are refined into a binary relationship. We discussed this type of refinement in detail in Section 3.6. It is important to note that in our notation,

Symbol	Meaning	
		Figure 3.14 Summary of the notation for ER diagrams.

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Indentifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
E_1 — R = E_2	Total Participation of E_2 in R
E_1 —1 R N— E_2	Cardinality Ratio 1:N for $E_1 : E_2$ in R
R (min, max) E	Structural Constraint (min, max) on Participation of E in R

once an attribute is replaced by a relationship, the attribute itself should be removed from the entity type to avoid duplication and redundancy.

- Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type. For example, suppose that each of several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE, has an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute Dept_name and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships. Other attributes/relationships of DEPARTMENT may be discovered later.

- An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute Dept_name and is related to only one other entity type, STUDENT. In this case, DEPARTMENT may be reduced or demoted to an attribute of STUDENT.

- Section 3.9 discusses choices concerning the degree of a relationship. In Chapter 4, we discuss other refinements concerning specialization/generalization.

3.7.4 Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. Appendix A gives some of the more popular notations. In Section 3.8, we introduce the Unified Modeling Language (UML) notation for class diagrams, which has been proposed as a standard for conceptual object modeling.

In this section, we describe one alternative ER notation for specifying structural constraints on relationships, which replaces the cardinality ratio (1:1, 1:N, M:N) and single/double-line notation for participation constraints. This notation involves associating a pair of integer numbers (min, max) with each *participation* of an entity type E in a relationship type R, where $0 \leq min \leq max$ and $max \geq 1$. The numbers mean that for each entity e in E, e must participate in at least *min* and at most *max* relationship instances in R *at any point in time*. In this method, min = 0 implies partial participation, whereas min > 0 implies total participation.

Figure 3.15 displays the COMPANY database schema using the (min, max) notation.[14] Usually, one uses either the cardinality ratio/single-line/double-line notation *or* the (min, max) notation. The (min, max) notation is more precise, and we can use it to specify some structural constraints for relationship types of *higher degree*. However, it is not sufficient for specifying some key constraints on higher-degree relationships, as discussed in Section 3.9.

Figure 3.15 also displays all the role names for the COMPANY database schema.

---

[14]In some notations, particularly those used in object modeling methodologies such as UML, the (min, max) is placed on the *opposite sides* to the ones we have shown. For example, for the WORKS_FOR relationship in Figure 3.15, the (1,1) would be on the DEPARTMENT side, and the (4,N) would be on the EMPLOYEE side. Here we used the original notation from Abrial (1974).
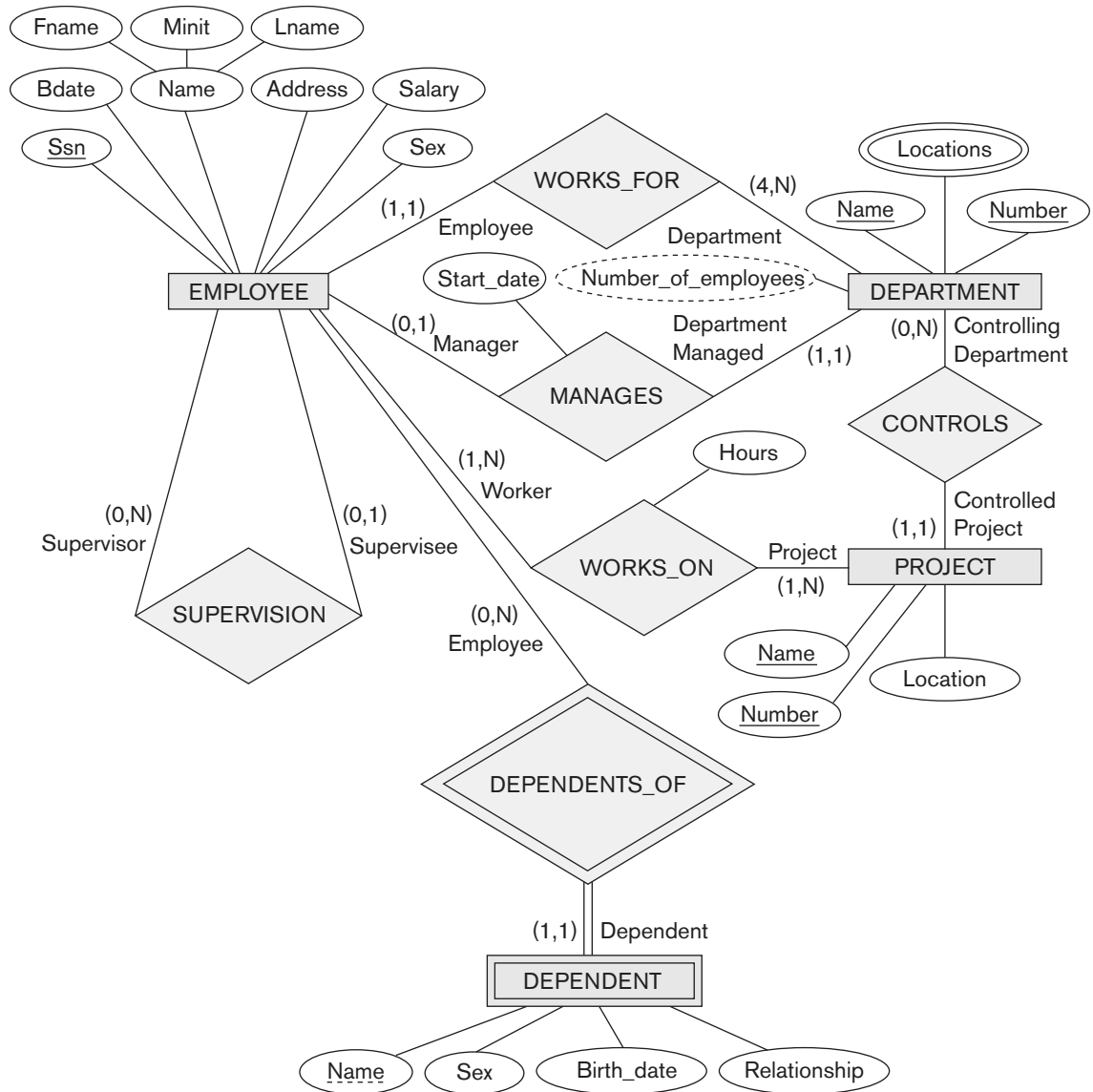
Figure 3.15
ER diagrams for the company schema, with structural constraints specified using
(min, max) notation and role names.

3.8 Example of Other Notation: UML Class Diagrams

The UML methodology is being used extensively in software design and has many
types of diagrams for various software design purposes. We only briefly present the
basics of **UML class diagrams** here and compare them with ER diagrams. In some
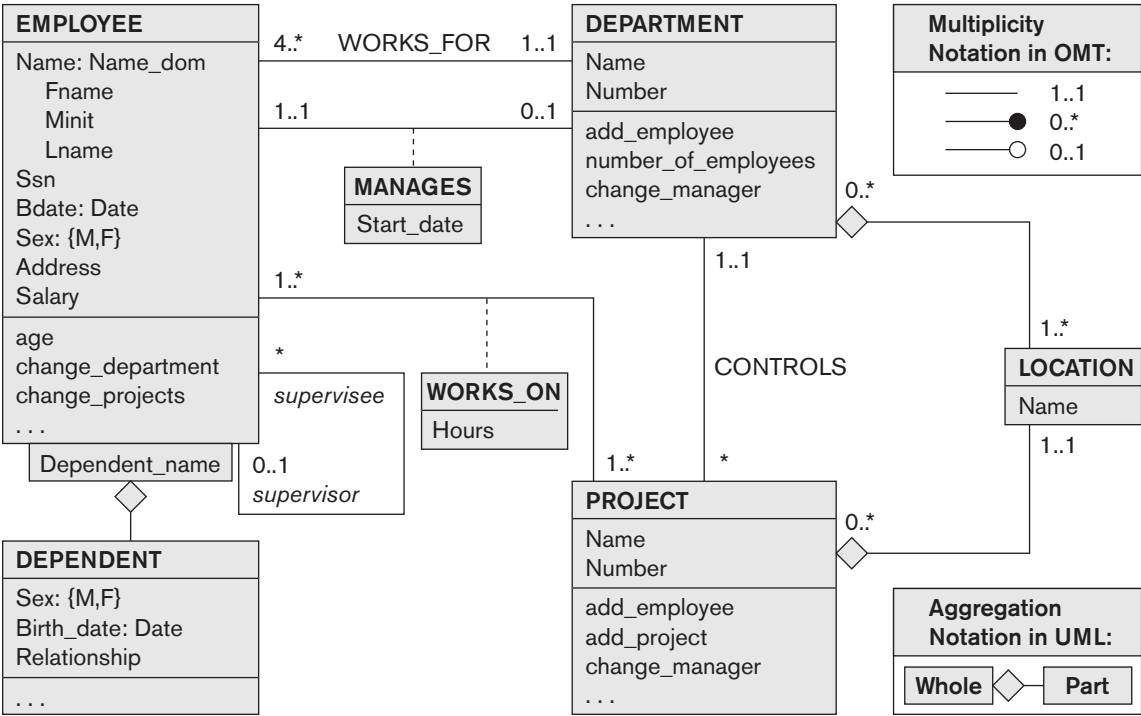
Figure 3.16
The COMPANY conceptual schema in UML class diagram notation.

ways, class diagrams can be considered as an alternative notation to ER diagrams. Additional UML notation and concepts are presented in Section 8.6. Figure 3.16 shows how the COMPANY ER database schema in Figure 3.15 can be displayed using UML class diagram notation. The *entity types* in Figure 3.15 are modeled as *classes* in Figure 3.16. An *entity* in ER corresponds to an *object* in UML.

In UML class diagrams, a **class** (similar to an entity type in ER) is displayed as a box (see Figure 3.16) that includes three sections: The top section gives the **class name** (similar to entity type name); the middle section includes the **attributes**; and the last section includes **operations** that can be applied to individual objects (similar to individual entities in an entity set) of the class. Operations are *not* specified in ER diagrams. Consider the EMPLOYEE class in Figure 3.16. Its attributes are Name, Ssn, Bdate, Sex, Address, and Salary. The designer can optionally specify the **domain** (or data type) of an attribute if desired, by placing a colon (:) followed by the domain name or description, as illustrated by the Name, Sex, and Bdate attributes of EMPLOYEE in Figure 3.16. A composite attribute is modeled as a **structured domain**, as illustrated by the Name attribute of EMPLOYEE. A multivalued attribute will generally be modeled as a separate class, as illustrated by the LOCATION class in Figure 3.16.

Relationship types are called **associations** in UML terminology, and relationship instances are called **links**. A **binary association** (binary relationship type) is represented as a line connecting the participating classes (entity types), and may optionally have a name. A relationship attribute, called a **link attribute**, is placed in a box that is connected to the association's line by a dashed line. The (min, max) notation described in Section 3.7.4 is used to specify relationship constraints, which are called **multiplicities** in UML terminology. Multiplicities are specified in the form *min..max*, and an asterisk (*) indicates no maximum limit on participation. However, the multiplicities are placed *on the opposite ends of the relationship* when compared with the (min, max) notation discussed in Section 3.7.4 (compare Figures 3.15 and 3.16). In UML, a single asterisk indicates a multiplicity of 0 ..*, and a single 1 indicates a multiplicity of 1..1. A recursive relationship type (see Section 3.4.2) is called a **reflexive association** in UML, and the role names—like the multiplicities—are placed at the opposite ends of an association when compared with the placing of role names in Figure 3.15.

In UML, there are two types of relationships: association and aggregation. **Aggregation** is meant to represent a relationship between a whole object and its component parts, and it has a distinct diagrammatic notation. In Figure 3.16, we modeled the locations of a department and the single location of a project as aggregations. However, aggregation and association do not have different structural properties, and the choice as to which type of relationship to use—aggregation or association—is somewhat subjective. In the ER model, both are represented as relationships.

UML also distinguishes between **unidirectional** and **bidirectional** associations (or aggregations). In the unidirectional case, the line connecting the classes is displayed with an arrow to indicate that only one direction for accessing related objects is needed. If no arrow is displayed, the bidirectional case is assumed, which is the default. For example, if we always expect to access the manager of a department starting from a DEPARTMENT object, we would draw the association line representing the MANAGES association with an arrow from DEPARTMENT to EMPLOYEE. In addition, relationship instances may be specified to be **ordered**. For example, we could specify that the employee objects related to each department through the WORKS_FOR association (relationship) should be ordered by their Start_date attribute value. Association (relationship) names are *optional* in UML, and relationship attributes are displayed in a box attached with a dashed line to the line representing the association/aggregation (see Start_date and Hours in Figure 3.16).

The operations given in each class are derived from the functional requirements of the application, as we discussed in Section 3.1. It is generally sufficient to specify the operation names initially for the logical operations that are expected to be applied to individual objects of a class, as shown in Figure 3.16. As the design is refined, more details are added, such as the exact argument types (parameters) for each operation, plus a functional description of each operation. UML has *function descriptions* and *sequence diagrams* to specify some of the operation details, but these are beyond the scope of our discussion.

Weak entities can be modeled using the UML construct called **qualified association** (or **qualified aggregation**); this can represent both the identifying relationship and the partial key, which is placed in a box attached to the owner class. This is illustrated by the DEPENDENT class and its qualified aggregation to EMPLOYEE in Figure 3.16. In UML terminology, the partial key attribute Dependent_name is called the **discriminator**, because its value distinguishes the objects associated with (related to) the same EMPLOYEE entity. Qualified associations are not restricted to modeling weak entities, and they can be used to model other situations in UML.

This section is not meant to be a complete description of UML class diagrams, but rather to illustrate one popular type of alternative diagrammatic notation that can be used for representing ER modeling concepts.

3.9 Relationship Types of Degree Higher than Two

In Section 3.4.2 we defined the **degree** of a relationship type as the number of participating entity types and called a relationship type of degree two *binary* and a relationship type of degree three *ternary*. In this section, we elaborate on the differences between binary and higher-degree relationships, when to choose higher-degree versus binary relationships, and how to specify constraints on higher-degree relationships.

3.9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

The ER diagram notation for a ternary relationship type is shown in Figure 3.17(a), which displays the schema for the SUPPLY relationship type that was displayed at the instance level in Figure 3.10. Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where the meaning is that s is a SUPPLIER who is currently supplying a PART p to a PROJECT j. In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

Figure 3.17(b) shows an ER diagram for three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s, p) whenever supplier s *can supply* part p (to any project); USES, between PROJECT and PART, includes an instance (j, p) whenever project j uses part p; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s supplies *some part* to project j. The existence of three relationship instances (s, p), (j, p), and (s, j) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (s, j, p) exists in the ternary relationship SUPPLY, because the *meaning is different*. It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree n or should be
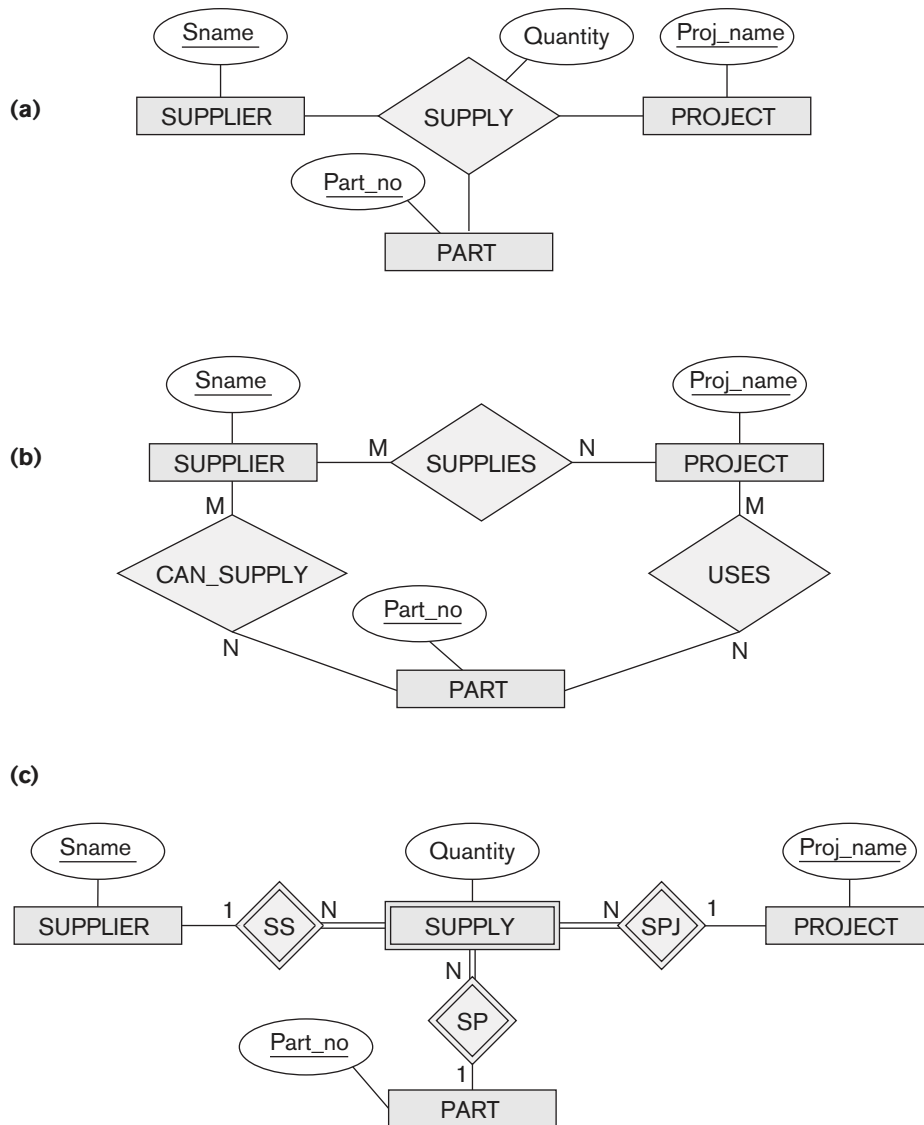
Figure 3.17
Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not
equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

broken down into several relationship types of smaller degrees. The designer must
base this decision on the semantics or meaning of the particular situation being
represented. The typical solution is to include the ternary relationship *plus* one or
more of the binary relationships, if they represent different meanings and if all are
needed by the application.
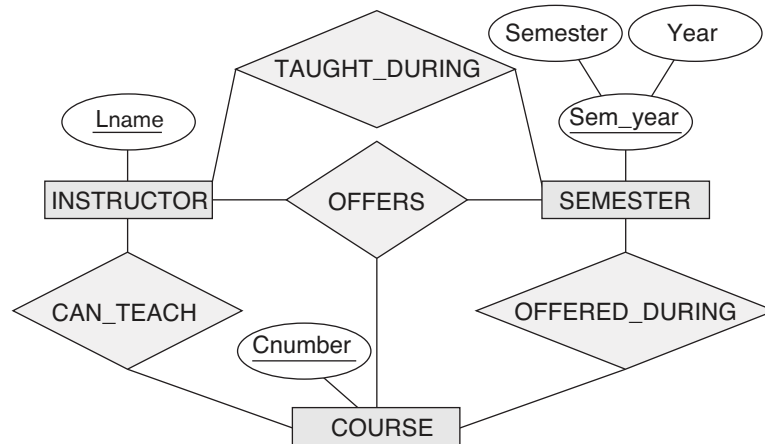
Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships. The three participating entity types SUPPLIER, PART, and PROJECT are together the owner entity types (see Figure 3.17(c)). Hence, an entity in the weak entity type SUPPLY in Figure 3.17(c) is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

It is also possible to represent the ternary relationship as a regular entity type by introducing an artificial or surrogate key. In this example, a key attribute Supply_id could be used for the supply entity type, converting it into a regular entity type. Three binary N:1 relationships relate SUPPLY to each of the three participating entity types.

Another example is shown in Figure 3.18. The ternary relationship type OFFERS represents information on instructors offering courses during particular semesters; hence it includes a relationship instance (*i*, *s*, *c*) whenever INSTRUCTOR *i* offers COURSE *c* during SEMESTER *s*. The three binary relationship types shown in Figure 3.18 have the following meanings: CAN_TEACH relates a course to the instructors who *can teach* that course, TAUGHT_DURING relates a semester to the instructors who *taught some course* during that semester, and OFFERED_DURING relates a semester to the courses offered during that semester *by any instructor*. These ternary and binary relationships represent different information, but certain constraints should hold among the relationships. For example, a relationship instance (*i*, *s*, *c*) should not exist in OFFERS *unless* an instance (*i*, *s*) exists in TAUGHT_DURING, an instance (*s*, *c*) exists in OFFERED_DURING, and an instance (*i*, *c*) exists in CAN_TEACH. However, the reverse is not always true; we may have instances (*i*, *s*), (*s*, *c*), and (*i*, *c*) in the three binary relationship types with no corresponding instance (*i*, *s*, *c*) in OFFERS. Note that in this example, based on the meanings of the relationships, we can infer the instances of TAUGHT_DURING and OFFERED_DURING from the instances in OFFERS, but
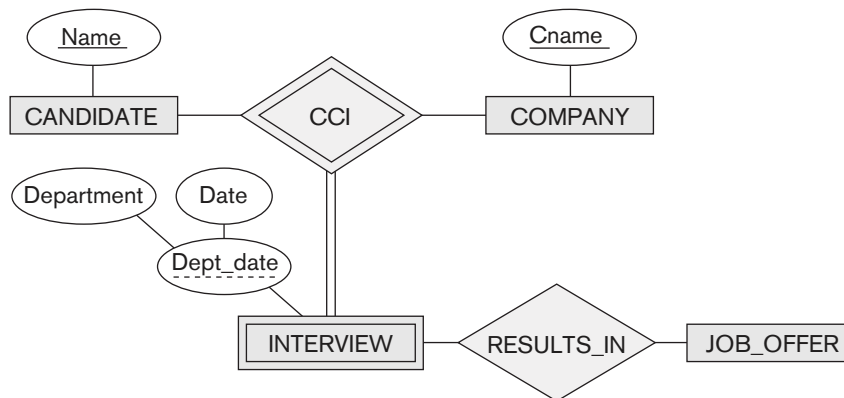
Figure 3.19
A weak entity type INTERVIEW with a ternary identifying relationship type.

we cannot infer the instances of CAN_TEACH; therefore, TAUGHT_DURING and OFFERED_DURING are redundant and can be left out.

Although in general three binary relationships *cannot* replace a ternary relationship, they may do so under certain *additional constraints*. In our example, if the CAN_TEACH relationship is 1:1 (an instructor can teach only one course, and a course can be taught by only one instructor), then the ternary relationship OFFERS can be left out because it can be inferred from the three binary relationships CAN_TEACH, TAUGHT_DURING, and OFFERED_DURING. The schema designer must analyze the meaning of each specific situation to decide which of the binary and ternary relationship types are needed.

Notice that it is possible to have a weak entity type with a ternary (or *n*-ary) identifying relationship type. In this case, the weak entity type can have *several* owner entity types. An example is shown in Figure 3.19. This example shows part of a database that keeps track of candidates interviewing for jobs at various companies, which may be part of an employment agency database. In the requirements, a candidate can have multiple interviews with the same company (for example, with different company departments or on separate dates), but a job offer is made based on one of the interviews. Here, INTERVIEW is represented as a weak entity with two owners CANDIDATE and COMPANY, and with the partial key Dept_date. An INTERVIEW entity is uniquely identified by a candidate, a company, and the combination of the date and department of the interview.

3.9.2 Constraints on Ternary (or Higher-Degree) Relationships

There are two notations for specifying structural constraints on *n*-ary relationships, and they specify different constraints. They should thus *both be used* if it is important to fully specify the structural constraints on a ternary or higher-degree relationship. The first notation is based on the cardinality ratio notation of binary relationships displayed in Figure 3.2. Here, a 1, M, or N is specified on each

participation arc (both M and N symbols stand for *many* or *any number*).[15] Let us illustrate this constraint using the SUPPLY relationship in Figure 3.17.

Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where s is a SUPPLIER, j is a PROJECT, and p is a PART. Suppose that the constraint exists that for a particular project-part combination, only one supplier will be used (only one supplier supplies a particular part to a particular project). In this case, we place 1 on the SUPPLIER participation, and M, N on the PROJECT, PART participations in Figure 3.17. This specifies the constraint that a particular (j, p) combination can appear at most once in the relationship set because each such (PROJECT, PART) combination uniquely determines a single supplier. Hence, any relationship instance (s, j, p) is uniquely identified in the relationship set by its (j, p) combination, which makes (j, p) a key for the relationship set. In this notation, the participations that have a 1 specified on them are not required to be part of the identifying key for the relationship set.[16] If all three cardinalities are M or N, then the key will be the combination of all three participants.

The second notation is based on the (min, max) notation displayed in Figure 3.15 for binary relationships. A (min, max) on a participation here specifies that each entity is related to at least *min* and at most *max relationship instances* in the relationship set. These constraints have no bearing on determining the key of an *n*-ary relationship, where $n > 2$,[17] but specify a different type of constraint that places restrictions on how many relationship instances each entity can participate in.

3.10 Another Example: A UNIVERSITY Database

We now present another example, a UNIVERSITY database, to illustrate the ER modeling concepts. Suppose that a database is needed to keep track of student enrollments in classes and students' final grades. After analyzing the miniworld rules and the users' needs, the requirements for this database were determined to be as follows (for brevity, we show the chosen entity type names and attribute names for the conceptual schema in parentheses as we describe the requirements; relationship type names are only shown in the ER schema diagram):

- The university is organized into colleges (COLLEGE), and each college has a unique name (CName), a main office (COffice) and phone (CPhone), and a particular faculty member who is dean of the college. Each college administers a number of academic departments (DEPT). Each department has a unique name (DName), a unique code number (DCode), a main office (DOffice) and phone (DPhone), and a particular faculty member who chairs the department. We keep track of the start date (CStartDate) when that faculty member began chairing the department.

---

[15]This notation allows us to determine the key of the *relationship relation*, as we discuss in Chapter 9.

[16]This is also true for cardinality ratios of binary relationships.

[17]The (min, max) constraints can determine the keys for binary relationships.

- A department offers a number of courses (COURSE), each of which has a unique course name (CoName), a unique code number (CCode), a course level (Level: this can be coded as 1 for freshman level, 2 for sophomore, 3 for junior, 4 for senior, 5 for MS level, and 6 for PhD level), a course credit hours (Credits), and a course description (CDesc). The database also keeps track of instructors (INSTRUCTOR); and each instructor has a unique identifier (Id), name (IName), office (IOffice), phone (IPhone), and rank (Rank); in addition, each instructor works for one primary academic department.

- The database will keep student data (STUDENT) and stores each student's name (SName, composed of first name (FName), middle name (MName), last name (LName)), student id (Sid, unique for every student), address (Addr), phone (Phone), major code (Major), and date of birth (DoB). A student is assigned to one primary academic department. It is required to keep track of the student's grades in each section the student has completed.

- Courses are offered as sections (SECTION). Each section is related to a single course and a single instructor and has a unique section identifier (SecId). A section also has a section number (SecNo: this is coded as 1, 2, 3, . . . for multiple sections offered during the same semester/year), semester (Sem), year (Year), classroom (CRoom: this is coded as a combination of building code (Bldg) and room number (RoomNo) within the building), and days/times (DaysTime: for example, 'MWF 9am-9.50am' or 'TR 3.30pm-5.20pm'— restricted to only allowed days/time values). (*Note*: The database will keep track of all the sections offered for the past several years, in addition to the current offerings. The SecId is unique for all sections, not just the sections for a particular semester.) The database keeps track of the students in each section, and the grade is recorded when available (this is a many-to-many relationship between students and sections). A section must have at least five students.

The ER diagram for these requirements is shown in Figure 3.20 using the min-max ER diagrammatic notation. Notice that for the SECTION entity type, we only showed SecID as an underlined key, but because of the miniworld constraints, several other combinations of values have to be unique for each section entity. For example, each of the following combinations must be unique based on the typical miniworld constraints:

1. (SecNo, Sem, Year, CCode (of the COURSE related to the SECTION)): This specifies that the section numbers of a particular course must be different during each particular semester and year.

2. (Sem, Year, CRoom, DaysTime): This specifies that in a particular semester and year, a classroom cannot be used by two different sections at the same days/time.

3. (Sem, Year, DaysTime, Id (of the INSTRUCTOR teaching the SECTION)): This specifies that in a particular semester and year, an instructor cannot teach two sections at the same days/time. Note that this rule will not apply if an instructor is allowed to teach two combined sections together in the particular university.

Can you think of any other attribute combinations that have to be unique?
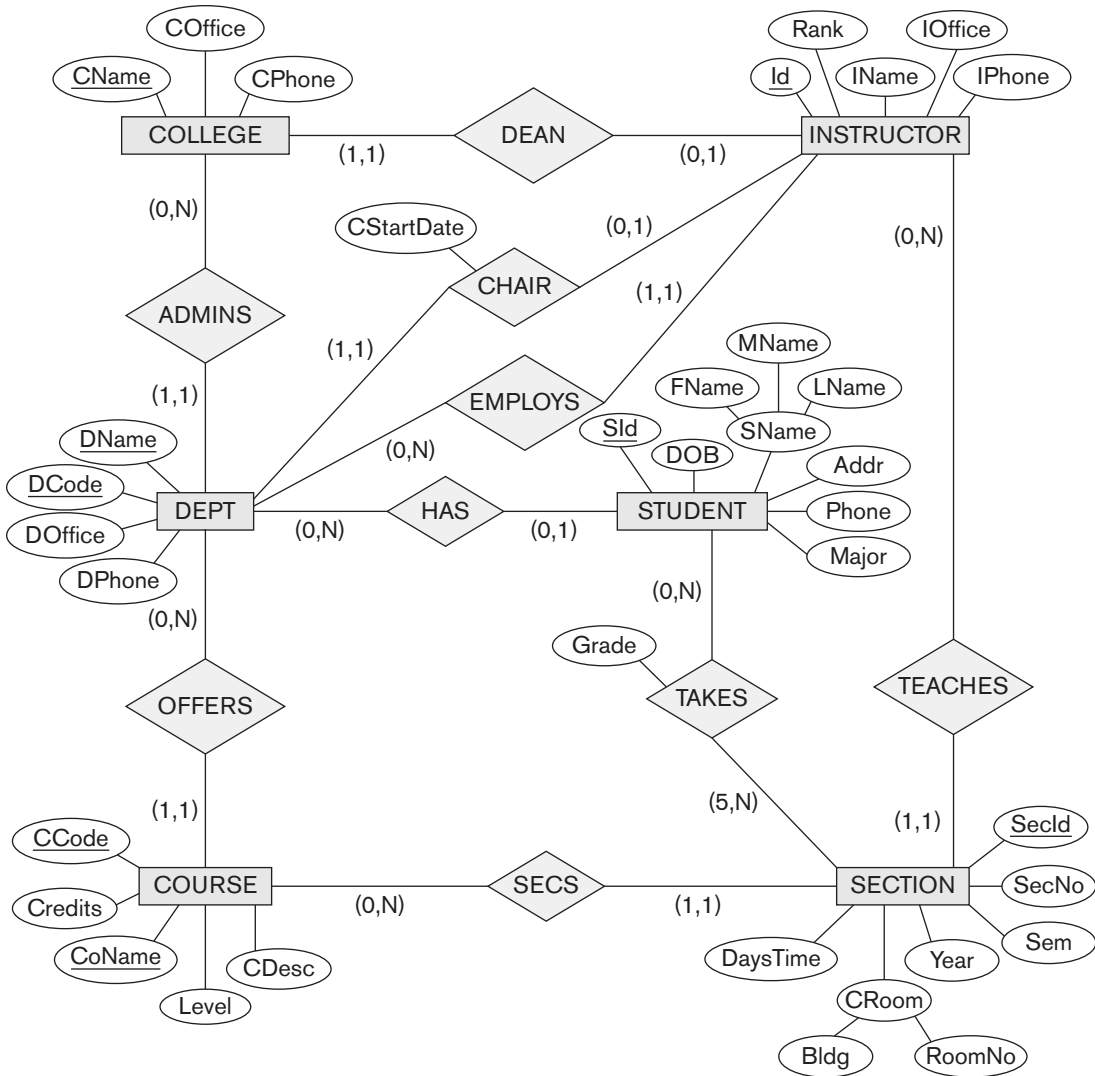
Figure 3.20
An ER diagram for a UNIVERSITY database schema.

3.11 Summary

In this chapter we presented the modeling concepts of a high-level conceptual data model, the entity–relationship (ER) model. We started by discussing the role that a high-level data model plays in the database design process, and then we presented a sample set of database requirements for the COMPANY database, which is one of the

examples that is used throughout this text. We defined the basic ER model concepts of entities and their attributes. Then we discussed NULL values and presented the various types of attributes, which can be nested arbitrarily to produce complex attributes:

- Simple or atomic
- Composite
- Multivalued

We also briefly discussed stored versus derived attributes. Then we discussed the ER model concepts at the schema or "intension" level:

- Entity types and their corresponding entity sets
- Key attributes of entity types
- Value sets (domains) of attributes
- Relationship types and their corresponding relationship sets
- Participation roles of entity types in relationship types

We presented two methods for specifying the structural constraints on relationship types. The first method distinguished two types of structural constraints:

- Cardinality ratios (1:1, 1:N, M:N for binary relationships)
- Participation constraints (total, partial)

We noted that, alternatively, another method of specifying structural constraints is to specify minimum and maximum numbers (min, max) on the participation of each entity type in a relationship type. We discussed weak entity types and the related concepts of owner entity types, identifying relationship types and partial key attributes.

Entity–relationship schemas can be represented diagrammatically as ER diagrams. We showed how to design an ER schema for the COMPANY database by first defining the entity types and their attributes and then refining the design to include relationship types. We displayed the ER diagram for the COMPANY database schema. We discussed some of the basic concepts of UML class diagrams and how they relate to ER modeling concepts. We also described ternary and higher-degree relationship types in more detail, and we discussed the circumstances under which they are distinguished from binary relationships. Finally, we presented requirements for a UNIVERSITY database schema as another example, and we showed the ER schema design.

The ER modeling concepts we have presented thus far—entity types, relationship types, attributes, keys, and structural constraints—can model many database applications. However, more complex applications—such as engineering design, medical information systems, and telecommunications—require additional concepts if we want to model them with greater accuracy. We discuss some advanced modeling concepts in Chapter 8 and revisit further advanced data modeling techniques in Chapter 26.

Review Questions

3.1. Discuss the role of a high-level data model in the database design process.

3.2. List the various cases where use of a NULL value would be appropriate.

3.3. Define the following terms: *entity, attribute, attribute value, relationship instance, composite attribute, multivalued attribute, derived attribute, complex attribute, key attribute,* and *value set* (*domain*).

3.4. What is an entity type? What is an entity set? Explain the differences among an entity, an entity type, and an entity set.

3.5. Explain the difference between an attribute and a value set.

3.6. What is a relationship type? Explain the differences among a relationship instance, a relationship type, and a relationship set.

3.7. What is a participation role? When is it necessary to use role names in the description of relationship types?

3.8. Describe the two alternatives for specifying structural constraints on relationship types. What are the advantages and disadvantages of each?

3.9. Under what conditions can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity types?

3.10. When we think of relationships as attributes, what are the value sets of these attributes? What class of data models is based on this concept?

3.11. What is meant by a recursive relationship type? Give some examples of recursive relationship types.

3.12. When is the concept of a weak entity used in data modeling? Define the terms *owner entity type, weak entity type, identifying relationship type,* and *partial key.*

3.13. Can an identifying relationship of a weak entity type be of a degree greater than two? Give examples to illustrate your answer.

3.14. Discuss the conventions for displaying an ER schema as an ER diagram.

3.15. Discuss the naming conventions used for ER schema diagrams.

Exercises

3.16. Which combinations of attributes have to be unique for each individual SECTION entity in the UNIVERSITY database shown in Figure 3.20 to enforce each of the following miniworld constraints:

   a. During a particular semester and year, only one section can use a particular classroom at a particular DaysTime value.

b. During a particular semester and year, an instructor can teach only one section at a particular DaysTime value.

c. During a particular semester and year, the section numbers for sections offered for the same course must all be different.

Can you think of any other similar constraints?

3.17. Composite and multivalued attributes can be nested to any number of levels. Suppose we want to design an attribute for a STUDENT entity type to keep track of previous college education. Such an attribute will have one entry for each college previously attended, and each such entry will be composed of college name, start and end dates, degree entries (degrees awarded at that college, if any), and transcript entries (courses completed at that college, if any). Each degree entry contains the degree name and the month and year the degree was awarded, and each transcript entry contains a course name, semester, year, and grade. Design an attribute to hold this information. Use the conventions in Figure 3.5.

3.18. Show an alternative design for the attribute described in Exercise 3.17 that uses only entity types (including weak entity types, if needed) and relationship types.

3.19. Consider the ER diagram in Figure 3.21, which shows a simplified schema for an airline reservations system. Extract from the ER diagram the requirements and constraints that produced this schema. Try to be as precise as possible in your requirements and constraints specification.

3.20. In Chapters 1 and 2, we discussed the database environment and database users. We can consider many entity types to describe such an environment, such as DBMS, stored database, DBA, and catalog/data dictionary. Try to specify all the entity types that can fully describe a database system and its environment; then specify the relationship types among them, and draw an ER diagram to describe such a general database environment.

3.21. Design an ER schema for keeping track of information about votes taken in the U.S. House of Representatives during the current two-year congressional session. The database needs to keep track of each U.S. STATE's Name (e.g., 'Texas', 'New York', 'California') and include the Region of the state (whose domain is {'Northeast', 'Midwest', 'Southeast', 'Southwest', 'West'}). Each CONGRESS_PERSON in the House of Representatives is described by his or her Name, plus the District represented, the Start_date when the congressperson was first elected, and the political Party to which he or she belongs (whose domain is {'Republican', 'Democrat', 'Independent', 'Other'}). The database keeps track of each BILL (i.e., proposed law), including the Bill_name, the Date_of_vote on the bill, whether the bill Passed_or_failed (whose domain is {'Yes', 'No'}), and the Sponsor (the congressperson(s) who sponsored—that is, proposed—the bill). The database also keeps track of how each congressperson voted on each bill (domain
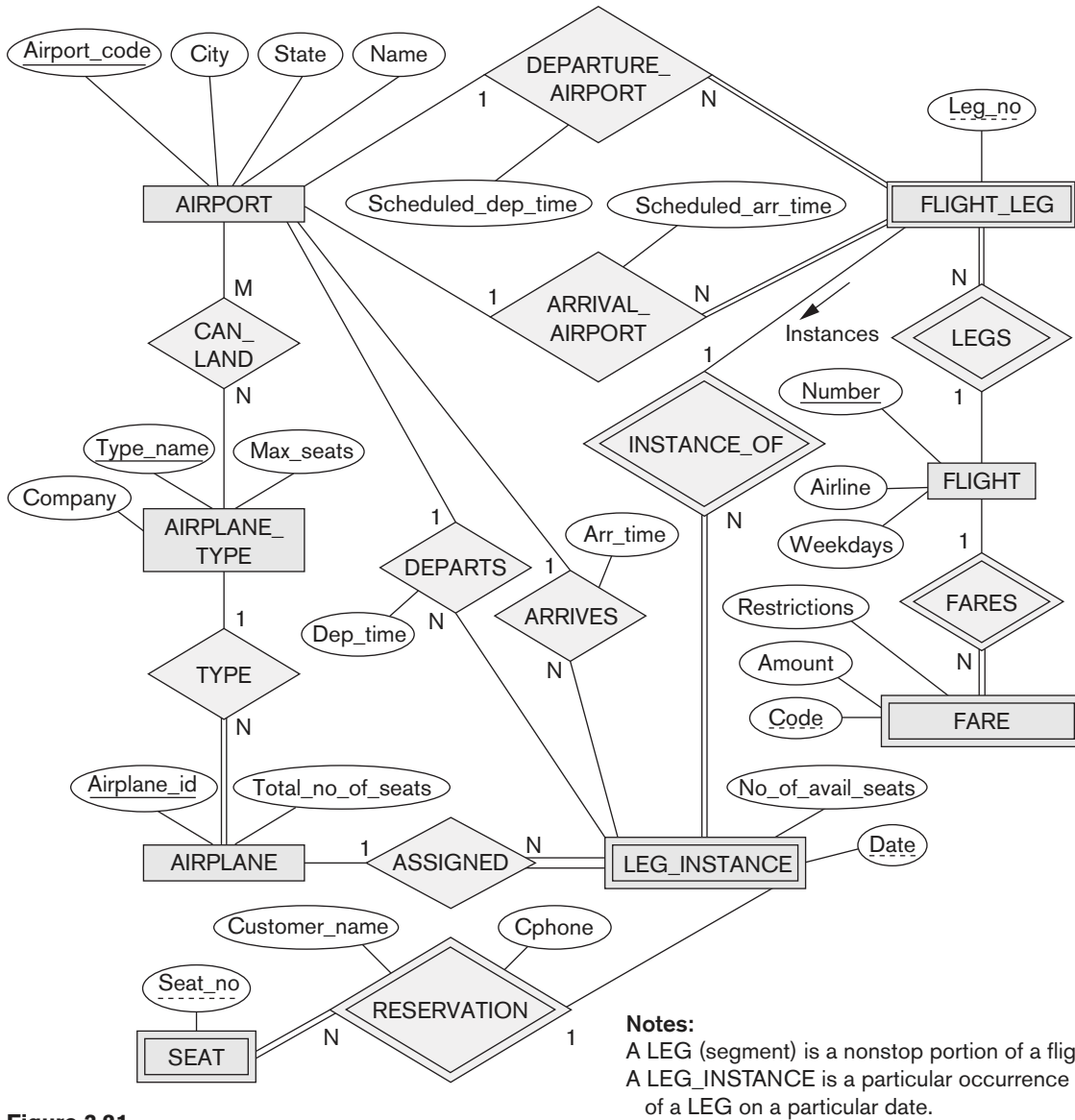
Figure 3.21
An ER diagram for an AIRLINE database schema.

of Vote attribute is {'Yes', 'No', 'Abstain', 'Absent'}). Draw an ER schema diagram for this application. State clearly any assumptions you make.

3.22. A database is being constructed to keep track of the teams and games of a sports league. A team has a number of players, not all of whom participate in each game. It is desired to keep track of the players participating in each game for each team, the positions they played in that game, and the result of

the game. Design an ER schema diagram for this application, stating any assumptions you make. Choose your favorite sport (e.g., soccer, baseball, football).

3.23. Consider the ER diagram shown in Figure 3.22 for part of a BANK database. Each bank can have multiple branches, and each branch can have multiple accounts and loans.

   a. List the strong (nonweak) entity types in the ER diagram.

   b. Is there a weak entity type? If so, give its name, partial key, and identifying relationship.

   c. What constraints do the partial key and the identifying relationship of the weak entity type specify in this diagram?

   d. List the names of all relationship types, and specify the (min, max) constraint on each participation of an entity type in a relationship type. Justify your choices.

Figure 3.22
An ER diagram for a BANK database schema.
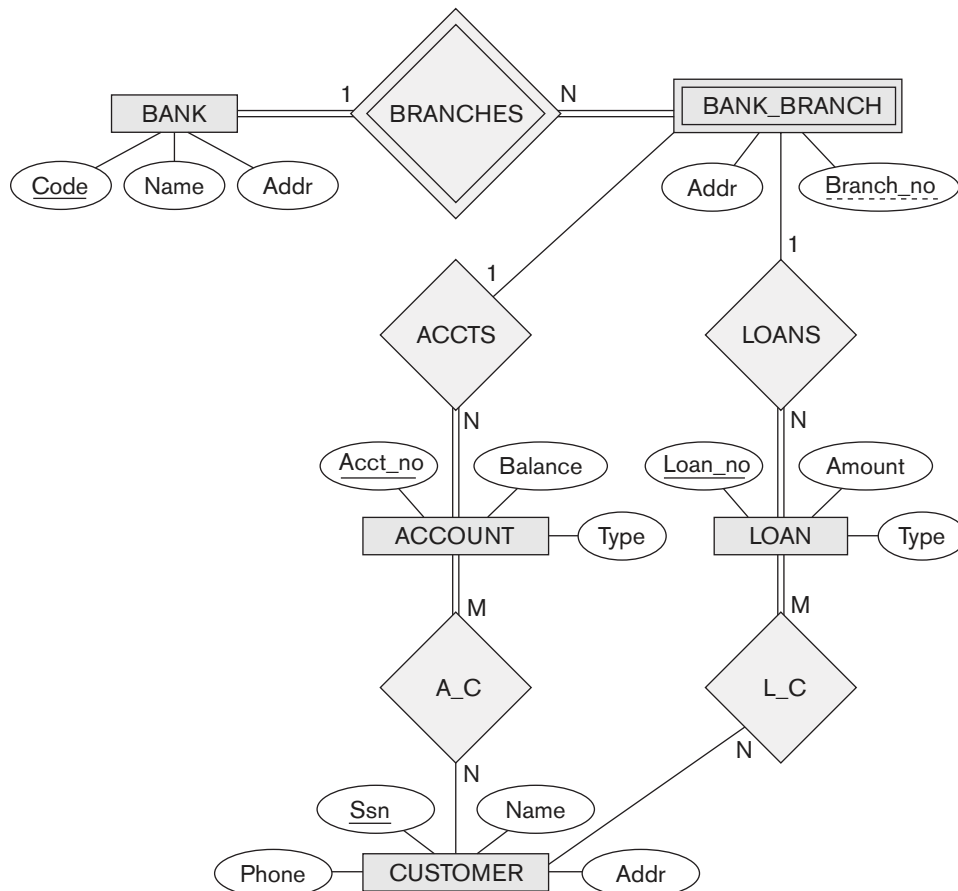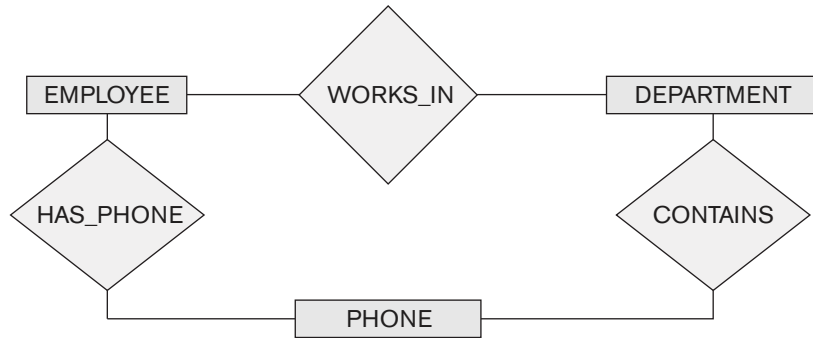
Figure 3.23
Part of an ER diagram
for a COMPANY
database.

e. List concisely the user requirements that led to this ER schema design.

f. Suppose that every customer must have at least one account but is restricted to at most two loans at a time, and that a bank branch cannot have more than 1,000 loans. How does this show up on the (min, max) constraints?

3.24. Consider the ER diagram in Figure 3.23. Assume that an employee may work in up to two departments or may not be assigned to any department. Assume that each department must have one and may have up to three phone numbers. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* Under what conditions would the relationship HAS_PHONE be redundant in this example?

3.25. Consider the ER diagram in Figure 3.24. Assume that a course may or may not use a textbook, but that a text by definition is a book that is used in some course. A course may not use more than five books. Instructors teach from two to four courses. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* If we add the relationship ADOPTS, to indicate the textbook(s) that an instructor uses for a course, should it be a binary relationship between INSTRUCTOR and TEXT, or a ternary relationship among all three entity types? What (min, max) constraints would you put on the relationship? Why?

Figure 3.24
Part of an ER diagram
for a COURSES
database.
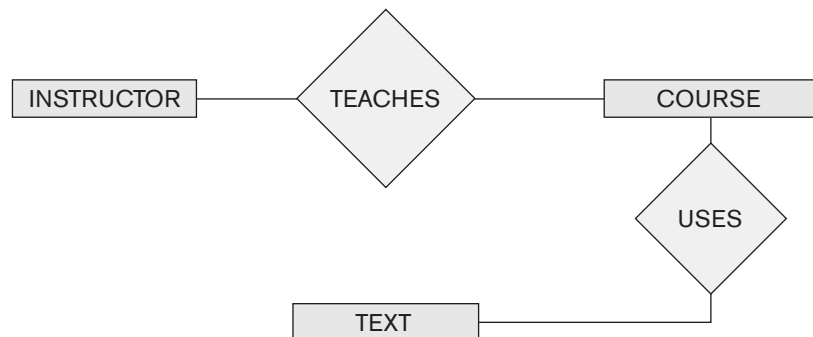
3.26. Consider an entity type SECTION in a UNIVERSITY database, which describes the section offerings of courses. The attributes of SECTION are Section_number, Semester, Year, Course_number, Instructor, Room_no (where section is taught), Building (where section is taught), Weekdays (domain is the possible combinations of weekdays in which a section can be offered {'MWF', 'MW', 'TT', and so on}), and Hours (domain is all possible time periods during which sections are offered {'9–9:50 A.M.', '10–10:50 A.M.', . . . , '3:30–4:50 P.M.', '5:30–6:20 P.M.', and so on}). Assume that Section_number is unique for each course within a particular semester/year combination (that is, if a course is offered multiple times during a particular semester, its section offerings are numbered 1, 2, 3, and so on). There are several composite keys for section, and some attributes are components of more than one key. Identify three composite keys, and show how they can be represented in an ER schema diagram.

3.27. Cardinality ratios often dictate the detailed design of a database. The cardinality ratio depends on the real-world meaning of the entity types involved and is defined by the specific application. For the following binary relationships, suggest cardinality ratios based on the common-sense meaning of the entity types. Clearly state any assumptions you make.

	Entity 1	Cardinality Ratio	Entity 2
1.	STUDENT	_____	SOCIAL_SECURITY_CARD
2.	STUDENT	_____	TEACHER
3.	CLASSROOM	_____	WALL
4.	COUNTRY	_____	CURRENT_PRESIDENT
5.	COURSE	_____	TEXTBOOK
6.	ITEM (that can be found in an order)	_____	ORDER
7.	STUDENT	_____	CLASS
8.	CLASS	_____	INSTRUCTOR
9.	INSTRUCTOR	_____	OFFICE
10.	EBAY_AUCTION_ITEM	_____	EBAY_BID

3.28. Consider the ER schema for the MOVIES database in Figure 3.25.

Assume that MOVIES is a populated database. ACTOR is used as a generic term and includes actresses. Given the constraints shown in the ER schema, respond to the following statements with *True, False*, or *Maybe*. Assign a response of *Maybe* to statements that, although not explicitly shown to be *True*, cannot be proven *False* based on the schema as shown. Justify each answer.

a. There are no actors in this database that have been in no movies.

b. There are some actors who have acted in more than ten movies.

c. Some actors have done a lead role in multiple movies.

d. A movie can have only a maximum of two lead actors.
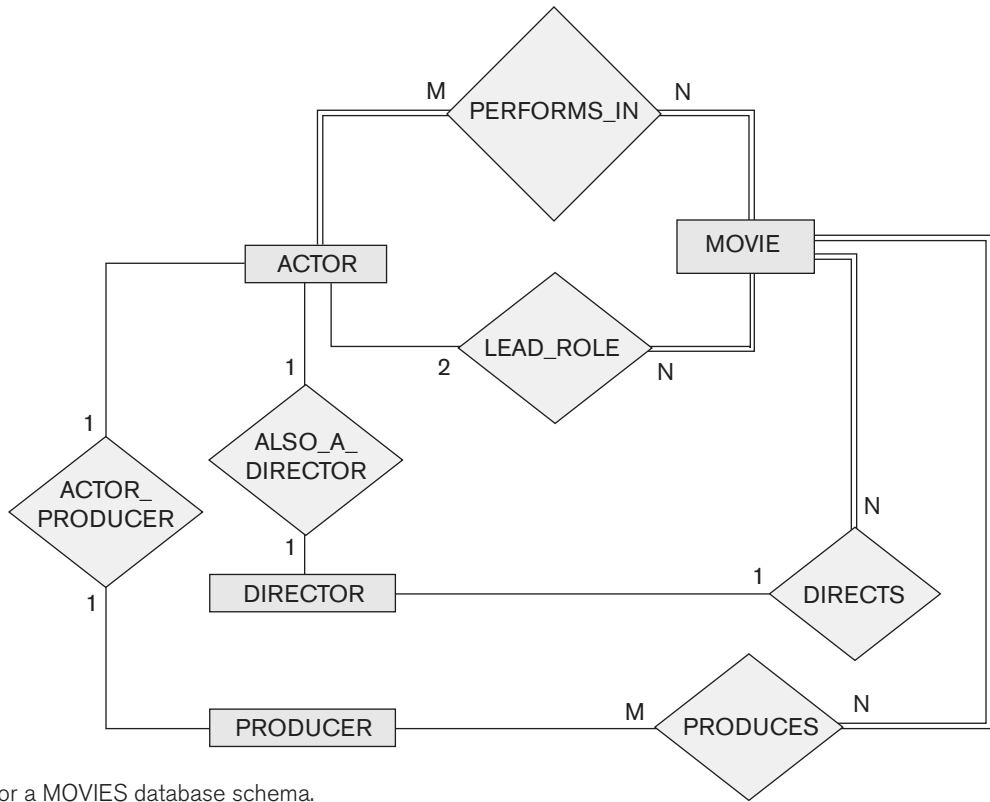
Figure 3.25
An ER diagram for a MOVIES database schema.

e. Every director has been an actor in some movie.

f. No producer has ever been an actor.

g. A producer cannot be an actor in some other movie.

h. There are movies with more than a dozen actors.

i. Some producers have been a director as well.

j. Most movies have one director and one producer.

k. Some movies have one director but several producers.

l. There are some actors who have done a lead role, directed a movie, and produced a movie.

m. No movie has a director who also acted in that movie.

3.29. Given the ER schema for the MOVIES database in Figure 3.25, draw an instance diagram using three movies that have been released recently. Draw instances of each entity type: MOVIES, ACTORS, PRODUCERS, DIRECTORS involved; make up instances of the relationships as they exist in reality for those movies.

3.30. Illustrate the UML diagram for Exercise 3.16. Your UML design should observe the following requirements:

  a. A student should have the ability to compute his/her GPA and add or drop majors and minors.

  b. Each department should be able to add or delete courses and hire or terminate faculty.

  c. Each instructor should be able to assign or change a student's grade for a course.

  Note: Some of these functions may be spread over multiple classes.

Laboratory Exercises

3.31. Consider the UNIVERSITY database described in Exercise 3.16. Build the ER schema for this database using a data modeling tool such as ERwin or Rational Rose.

3.32. Consider a MAIL_ORDER database in which employees take orders for parts from customers. The data requirements are summarized as follows:

  ◾ The mail order company has employees, each identified by a unique employee number, first and last name, and Zip Code.

  ◾ Each customer of the company is identified by a unique customer number, first and last name, and Zip Code.

  ◾ Each part sold by the company is identified by a unique part number, a part name, price, and quantity in stock.

  ◾ Each order placed by a customer is taken by an employee and is given a unique order number. Each order contains specified quantities of one or more parts. Each order has a date of receipt as well as an expected ship date. The actual ship date is also recorded.

  Design an entity–relationship diagram for the mail order database and build the design using a data modeling tool such as ERwin or Rational Rose.

3.33. Consider a MOVIE database in which data is recorded about the movie industry. The data requirements are summarized as follows:

  ◾ Each movie is identified by title and year of release. Each movie has a length in minutes. Each has a production company, and each is classified under one or more genres (such as horror, action, drama, and so forth). Each movie has one or more directors and one or more actors appear in it. Each movie also has a plot outline. Finally, each movie has zero or more quotable quotes, each of which is spoken by a particular actor appearing in the movie.

  ◾ Actors are identified by name and date of birth and appear in one or more movies. Each actor has a role in the movie.

- Directors are also identified by name and date of birth and direct one or more movies. It is possible for a director to act in a movie (including one that he or she may also direct).
- Production companies are identified by name and each has an address. A production company produces one or more movies.

Design an entity–relationship diagram for the movie database and enter the design using a data modeling tool such as ERwin or Rational Rose.

3.34. Consider a CONFERENCE_REVIEW database in which researchers submit their research papers for consideration. Reviews by reviewers are recorded for use in the paper selection process. The database system caters primarily to reviewers who record answers to evaluation questions for each paper they review and make recommendations regarding whether to accept or reject the paper. The data requirements are summarized as follows:

- Authors of papers are uniquely identified by e-mail id. First and last names are also recorded.
- Each paper is assigned a unique identifier by the system and is described by a title, abstract, and the name of the electronic file containing the paper.
- A paper may have multiple authors, but one of the authors is designated as the contact author.
- Reviewers of papers are uniquely identified by e-mail address. Each reviewer's first name, last name, phone number, affiliation, and topics of interest are also recorded.
- Each paper is assigned between two and four reviewers. A reviewer rates each paper assigned to him or her on a scale of 1 to 10 in four categories: technical merit, readability, originality, and relevance to the conference. Finally, each reviewer provides an overall recommendation regarding each paper.
- Each review contains two types of written comments: one to be seen by the review committee only and the other as feedback to the author(s).

Design an entity–relationship diagram for the CONFERENCE_REVIEW database and build the design using a data modeling tool such as ERwin or Rational Rose.

3.35. Consider the ER diagram for the AIRLINE database shown in Figure 3.21. Build this design using a data modeling tool such as ERwin or Rational Rose.

Selected Bibliography

The entity–relationship model was introduced by Chen (1976), and related work appears in Schmidt and Swenson (1975), Wiederhold and Elmasri (1979), and Senko (1975). Since then, numerous modifications to the ER model have been suggested. We have incorporated some of these in our presentation. Structural

constraints on relationships are discussed in Abrial (1974), Elmasri and Wieder-hold (1980), and Lenzerini and Santucci (1983). Multivalued and composite attributes are incorporated in the ER model in Elmasri et al. (1985). Although we did not discuss languages for the ER model and its extensions, there have been several proposals for such languages. Elmasri and Wiederhold (1981) proposed the GORDAS query language for the ER model. Another ER query language was proposed by Markowitz and Raz (1983). Senko (1980) presented a query language for Senko's DIAM model. A formal set of operations called the ER algebra was presented by Parent and Spaccapietra (1985). Gogolla and Hohenstein (1991) presented another formal language for the ER model. Campbell et al. (1985) presented a set of ER operations and showed that they are relationally complete. A conference for the dissemination of research results related to the ER model has been held regularly since 1979. The conference, now known as the International Conference on Conceptual Modeling, has been held in Los Angeles (ER 1979, ER 1983, ER 1997), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, France (ER 1986), New York City (ER 1987), Rome (ER 1988), Toronto (ER 1989), Lausanne, Switzerland (ER 1990), San Mateo, California (ER 1991), Karlsruhe, Germany (ER 1992), Arlington, Texas (ER 1993), Manchester, England (ER 1994), Brisbane, Australia (ER 1995), Cottbus, Germany (ER 1996), Singapore (ER 1998), Paris, France (ER 1999), Salt Lake City, Utah (ER 2000), Yokohama, Japan (ER 2001), Tampere, Finland (ER 2002), Chicago, Illinois (ER 2003), Shanghai, China (ER 2004), Klagenfurt, Austria (ER 2005), Tucson, Arizona (ER 2006), Auckland, New Zealand (ER 2007), Barcelona, Catalonia, Spain (ER 2008), and Gramado, RS, Brazil (ER 2009). The 2010 conference was held in Vancouver, British Columbia, Canada (ER2010), 2011 in Brussels, Belgium (ER2011), 2012 in Florence, Italy (ER2012) , 2013 in Hong Kong, China (ER2013), and the 2014 conference was held in Atlanta, Georgia (ER 2014). The 2015 conference is to be held in Stockholm, Sweden.

This page intentionally left blank

The Enhanced Entity–Relationship (EER) Model

The ER modeling concepts discussed in Chapter 3 are sufficient for representing many database schemas for *traditional* database applications, which include many data-processing applications in business and industry. Since the late 1970s, however, designers of database applications have tried to design more accurate database schemas that reflect the data properties and constraints more precisely. This was particularly important for newer applications of database technology, such as databases for engineering design and manufacturing (CAD/CAM),[1] telecommunications, complex software systems, and geographic information systems (GISs), among many other applications. These types of databases have requirements that are more complex than the more traditional applications. This led to the development of additional *semantic data modeling* concepts that were incorporated into conceptual data models such as the ER model. Various semantic data models have been proposed in the literature. Many of these concepts were also developed independently in related areas of computer science, such as the **knowledge representation** area of artificial intelligence and the **object modeling** area in software engineering.

In this chapter, we describe features that have been proposed for semantic data models and show how the ER model can be enhanced to include these concepts, which leads to the **enhanced ER (EER)** model.[2] We start in Section 4.1 by incorporating the concepts of *class/subclass relationships* and *type inheritance* into the ER model. Then, in Section 4.2, we add the concepts of *specialization* and *generalization.* Section 4.3 discusses the various types of *constraints* on specialization/generalization, and Section 4.4 shows how the UNION construct can be modeled by including the

---

[1]CAD/CAM stands for computer-aided design/computer-aided manufacturing.

[2]EER has also been used to stand for *extended* ER model.

concept of *category* in the EER model. Section 4.5 gives a sample UNIVERSITY database schema in the EER model and summarizes the EER model concepts by giving formal definitions. We will use the terms *object* and *entity* interchangeably in this chapter, because many of these concepts are commonly used in object-oriented models.

We present the UML class diagram notation for representing specialization and generalization in Section 4.6, and we briefly compare these with EER notation and concepts. This serves as an example of alternative notation, and is a continuation of Section 3.8, which presented basic UML class diagram notation that corresponds to the basic ER model. In Section 4.7, we discuss the fundamental abstractions that are used as the basis of many semantic data models. Section 4.8 summarizes the chapter.

For a detailed introduction to conceptual modeling, Chapter 4 should be considered a continuation of Chapter 3. However, if only a basic introduction to ER modeling is desired, this chapter may be omitted. Alternatively, the reader may choose to skip some or all of the later sections of this chapter (Sections 4.4 through 4.8).

4.1 Subclasses, Superclasses, and Inheritance

The EER model includes *all the modeling concepts of the ER model* that were presented in Chapter 3. In addition, it includes the concepts of **subclass** and **superclass** and the related concepts of **specialization** and **generalization** (see Sections 4.2 and 4.3). Another concept included in the EER model is that of a **category** or **union type** (see Section 4.4), which is used to represent a collection of objects (entities) that is the *union* of objects of different entity types. Associated with these concepts is the important mechanism of **attribute and relationship inheritance**. Unfortunately, no standard terminology exists for these concepts, so we use the most common terminology. Alternative terminology is given in footnotes. We also describe a diagrammatic technique for displaying these concepts when they arise in an EER schema. We call the resulting schema diagrams **enhanced ER** or **EER diagrams**.

The first enhanced ER (EER) model concept we take up is that of a **subtype** or **subclass** of an entity type. As we discussed in Chapter 3, the name of an entity type is used to represent both a *type of entity* and the *entity set* or *collection of entities of that type* that exist in the database. For example, the entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database. In many cases an entity type has numerous subgroupings or subtypes of its entities that are meaningful and need to be represented explicitly because of their significance to the database application. For example, the entities that are members of the EMPLOYEE entity type may be distinguished further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on. The set or collection of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these subgroupings is also an employee. We call each of these subgroupings a

subclass or **subtype** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** or **supertype** for each of these subclasses. Figure 4.1 shows how to represent these concepts diagramatically in EER diagrams. (The circle notation in Figure 4.1 will be explained in Section 4.2.)

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or **supertype/subtype** or simply **class/subclass relationship**.[3] In our previous example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships. Notice that a member entity of the subclass represents the *same real-world entity* as some member of the superclass; for example, a SECRETARY entity 'Joan Logano' is also the EMPLOYEE 'Joan Logano.' Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role.* When we implement a superclass/subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity. In Section 9.2, we discuss various options for representing superclass/subclass relationships in relational databases.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally
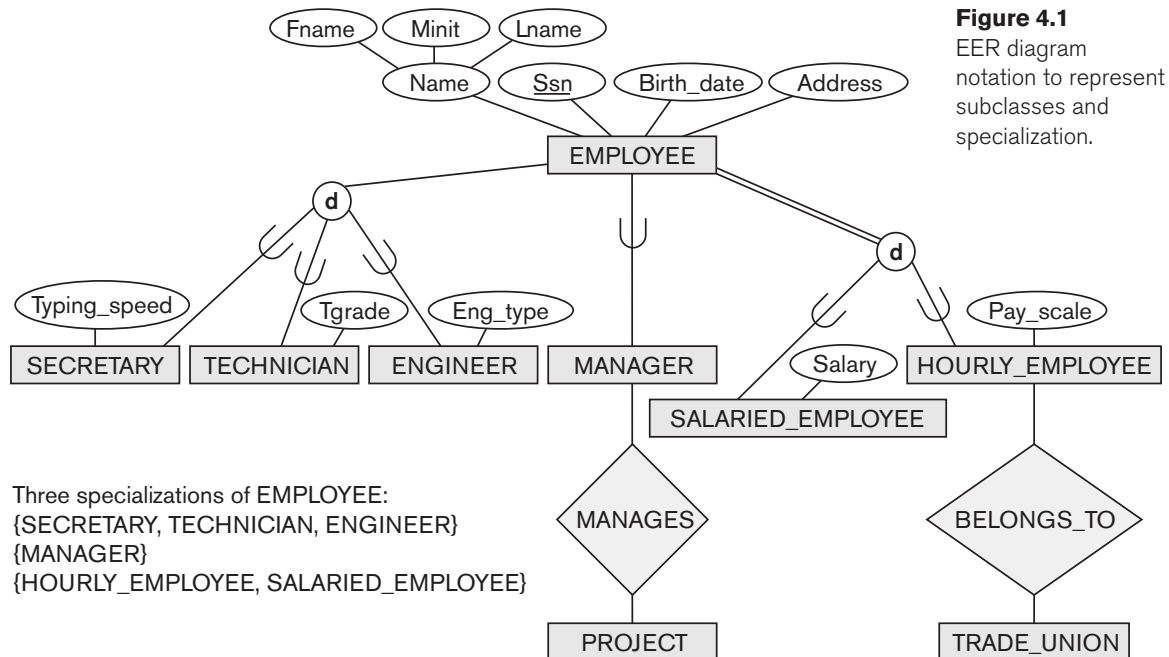
Figure 4.1
EER diagram notation to represent subclasses and specialization.

Three specializations of EMPLOYEE:
{SECRETARY, TECHNICIAN, ENGINEER}
{MANAGER}
{HOURLY_EMPLOYEE, SALARIED_EMPLOYEE}

---

[3]A class/subclass relationship is often called an **IS-A** (or **IS-AN**) **relationship** because of the way we refer to the concept. We say a SECRETARY *is an* EMPLOYEE, a TECHNICIAN *is an* EMPLOYEE, and so on.

as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE of the EMPLOYEE entity type. However, it is not necessary that every entity in a superclass is a member of some subclass.

An important concept associated with subclasses (subtypes) is that of **type inheritance**. Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes *as well as* values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass **inherits** all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates. Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass, can be considered an *entity type* in its own right.[4]

4.2 Specialization and Generalization

4.2.1 Specialization

Specialization is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the *job type* of each employee. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the *method of pay*.

Figure 4.1 shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship.[5] Attributes that apply only to entities of a particular subclass—such as TypingSpeed of SECRETARY—are attached to the rectangle representing that subclass. These are called **specific** (or **local**) **attributes** of the subclass. Similarly, a subclass can participate in **specific relationship types**, such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO

---

[4]In some object-oriented programming languages, a common restriction is that an entity (or object) has *only one type*. This is generally too restrictive for conceptual database modeling.

[5]There are many alternative notations for specialization; we present the UML notation in Section 4.6 and other proposed notations in Appendix A.

relationship in Figure 4.1. We will explain the **d** symbol in the circles in Figure 4.1 and additional EER diagram notation shortly.

Figure 4.2 shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, e_1 is shown in both EMPLOYEE and SECRETARY in Figure 4.2. As the figure suggests, a superclass/subclass relationship such as EMPLOYEE/SECRETARY somewhat resembles a 1:1 relationship *at the instance level* (see Figure 3.12). The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a superclass/subclass relationship the entity in the subclass is the same real-world entity as the entity in the superclass but is playing a *specialized role*—for example, an EMPLOYEE specialized in the role of SECRETARY, or an EMPLOYEE specialized in the role of TECHNICIAN.

There are two main reasons for including class/subclass relationships and specializations. The first is that certain attributes may apply to some but not all entities of
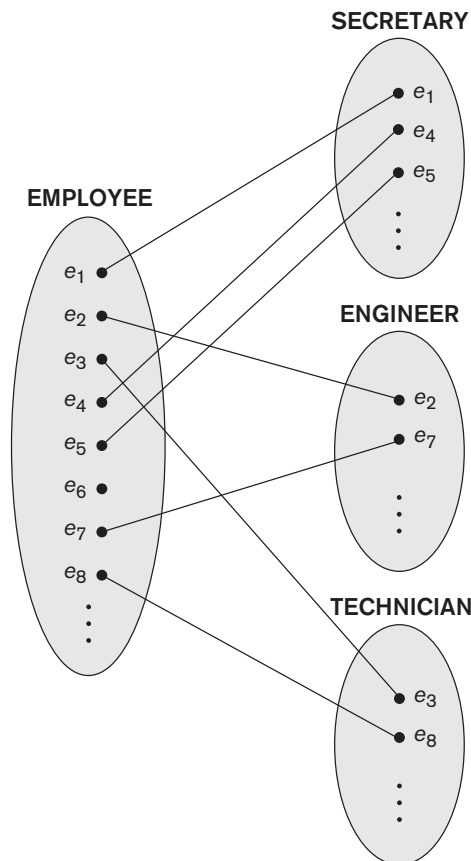
Figure 4.2
Instances of a specialization.

the superclass entity type. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, in Figure 4.1 the SECRETARY subclass has the specific attribute Typing_speed, whereas the ENGINEER subclass has the specific attribute Eng_type, but SECRETARY and ENGINEER share their other inherited attributes from the EMPLOYEE entity type.

The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Figure 4.1.

4.2.2 Generalization

We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses**. For example, consider the entity types CAR and TRUCK shown in Figure 4.3(a). Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Figure 4.3(b). Both CAR and TRUCK are now subclasses of the

Figure 4.3
Generalization. (a) Two entity types, CAR and TRUCK.
(b) Generalizing CAR and TRUCK into the superclass VEHICLE.
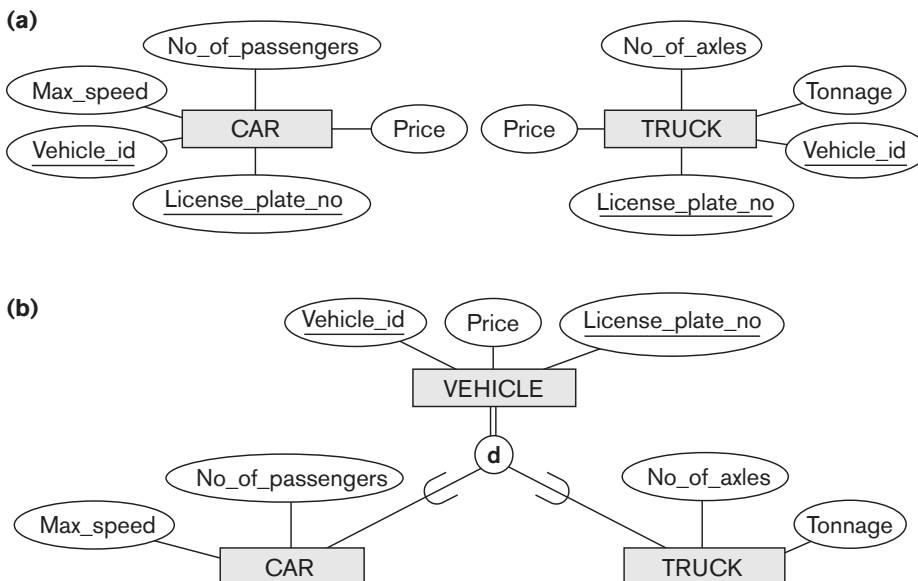
generalized superclass VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process; we can view {CAR, TRUCK} as a specialization of VEHICLE rather than viewing VEHICLE as a generalization of CAR and TRUCK. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization process, whereas arrows pointing to the specialized subclasses represent a specialization process. We will *not* use this notation because the decision as to which process was followed in a particular situation is often subjective.

So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type*; that is why superclasses and subclasses are all shown in rectangles in EER diagrams, like entity types.

4.3 Constraints and Characteristics of Specialization and Generalization Hierarchies

First, we discuss constraints that apply to a single specialization or a single generalization. For brevity, our discussion refers only to *specialization* even though it applies to *both* specialization and generalization. Then, we discuss differences between specialization/generalization *lattices* (*multiple inheritance*) and *hierarchies* (*single inheritance*), and we elaborate on the differences between the specialization and generalization processes during conceptual database schema design.

4.3.1 Constraints on Specialization and Generalization

In general, we may have several specializations defined on the same entity type (or superclass), as shown in Figure 4.1. In such a case, entities may belong to subclasses in each of the specializations. A specialization may also consist of a *single* subclass only, such as the {MANAGER} specialization in Figure 4.1; in such a case, we do not use the circle notation.

In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called **predicate-defined** (or **condition-defined**) **subclasses**. For example, if the EMPLOYEE entity type has an attribute Job_type, as shown in Figure 4.4, we can specify the condition of membership in the SECRETARY subclass by the condition (Job_type = 'Secretary'), which we call the **defining predicate** of the subclass. This condition is a *constraint* specifying that exactly those entities of the EMPLOYEE entity type whose attribute value for Job_type
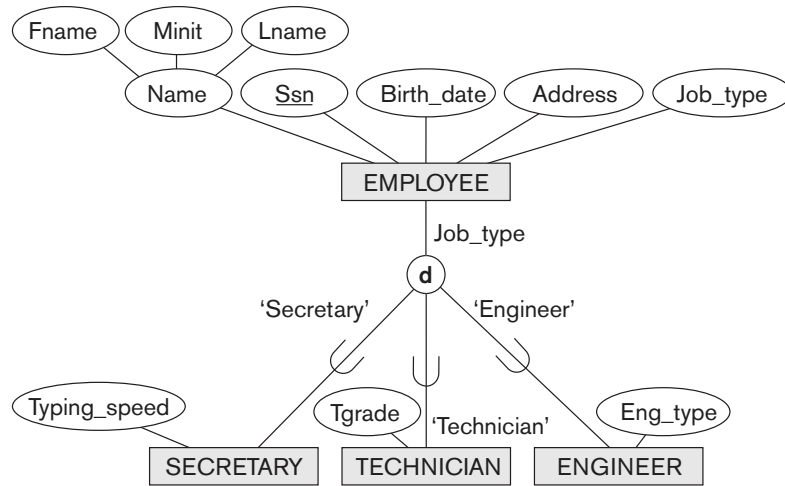
Figure 4.4
EER diagram notation for an attribute-defined specialization on Job_type.

is 'Secretary' belong to the subclass. We display a predicate-defined subclass by writing the predicate condition next to the line that connects the subclass to the specialization circle.

If *all* subclasses in a specialization have their membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization**, and the attribute is called the **defining attribute** of the specialization.[6] In this case, all the entities with the same value for the attribute belong to the same subclass. We display an attribute-defined specialization by placing the defining attribute name next to the arc from the circle to the superclass, as shown in Figure 4.4.

When we do not have a condition for determining membership in a subclass, the subclass is called **user-defined**. Membership in such a subclass is determined by the database users when they apply the operation to add an entity to the subclass; hence, membership is *specified individually for each entity by the user*, not by any condition that may be evaluated automatically.

Two other constraints may apply to a specialization. The first is the **disjointness constraint**, which specifies that the subclasses of the specialization must be disjoint sets. This means that an entity can be a member of *at most* one of the subclasses of the specialization. A specialization that is attribute-defined implies the disjointness constraint (if the attribute used to define the membership predicate is single-valued). Figure 4.4 illustrates this case, where the **d** in the circle stands for *disjoint*. The **d** notation also applies to user-defined subclasses of a specialization that must be disjoint, as illustrated by the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 4.1. If the subclasses are not constrained to be disjoint, their sets of entities

---

[6]Such an attribute is called a *discriminator* or *discriminating attribute* in UML terminology.
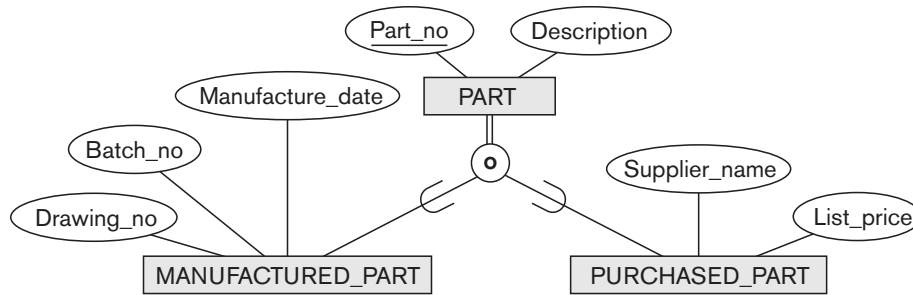
Figure 4.5
EER diagram notation for an overlapping (nondisjoint) specialization.

may be **overlapping**; that is, the same (real-world) entity may be a member of more than one subclass of the specialization. This case, which is the default, is displayed by placing an **o** in the circle, as shown in Figure 4.5.

The second constraint on specialization is called the **completeness** (or **totalness**) **constraint**, which may be total or partial. A **total specialization** constraint specifies that *every* entity in the superclass must be a member of at least one subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 4.1 is a total specialization of EMPLOYEE. This is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a **partial specialization**, which allows an entity not to belong to any of the subclasses. For example, if some EMPLOYEE entities do not belong to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} in Figures 4.1 and 4.4, then that specialization is partial.[7]

Notice that the disjointness and completeness constraints are *independent.* Hence, we have the following four possible constraints on a specialization:

- Disjoint, total
- Disjoint, partial
- Overlapping, total
- Overlapping, partial

Of course, the correct constraint is determined from the real-world meaning that applies to each specialization. In general, a superclass that was identified through the *generalization* process usually is **total**, because the superclass is *derived from* the subclasses and hence contains only the entities that are in the subclasses.

Certain insertion and deletion rules apply to specialization (and generalization) as a consequence of the constraints specified earlier. Some of these rules are as follows:

- Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.

---

[7]The notation of using single or double lines is similar to that for partial or total participation of an entity type in a relationship type, as described in Chapter 3.

- Inserting an entity in a superclass implies that the entity is mandatorily inserted in all *predicate-defined* (or *attribute-defined*) subclasses for which the entity satisfies the defining predicate.
- Inserting an entity in a superclass of a *total specialization* implies that the entity is mandatorily inserted in at least one of the subclasses of the specialization.

The reader is encouraged to make a complete list of rules for insertions and deletions for the various types of specializations.

4.3.2 Specialization and Generalization Hierarchies and Lattices

A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specializations. For example, in Figure 4.6 ENGINEER is a subclass of EMPLOYEE and is also a superclass of ENGINEERING_MANAGER; this represents the real-world constraint that every engineering manager is required to be an engineer. A **specialization hierarchy** has the constraint that every subclass participates *as a subclass* in *only one* class/subclass relationship; that is, each subclass has only one parent, which results in a **tree structure** or **strict hierarchy**. In contrast, for a **specialization lattice**, a subclass can be a subclass in *more than one* class/subclass relationship. Hence, Figure 4.6 is a lattice.

Figure 4.7 shows another specialization lattice of more than one level. This may be part of a conceptual schema for a UNIVERSITY database. Notice that this arrangement would have been a hierarchy except for the STUDENT_ASSISTANT subclass, which is a subclass in two distinct class/subclass relationships.

Figure 4.6
A specialization lattice with shared subclass ENGINEERING_MANAGER.
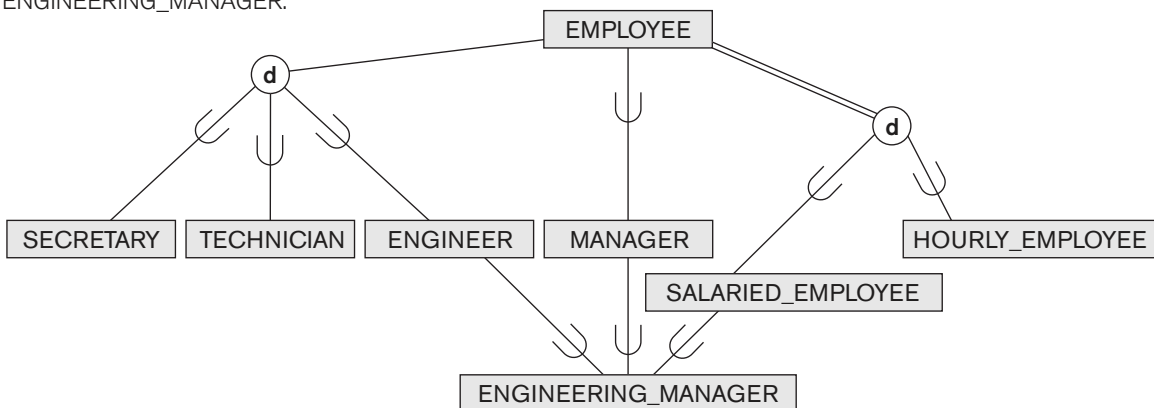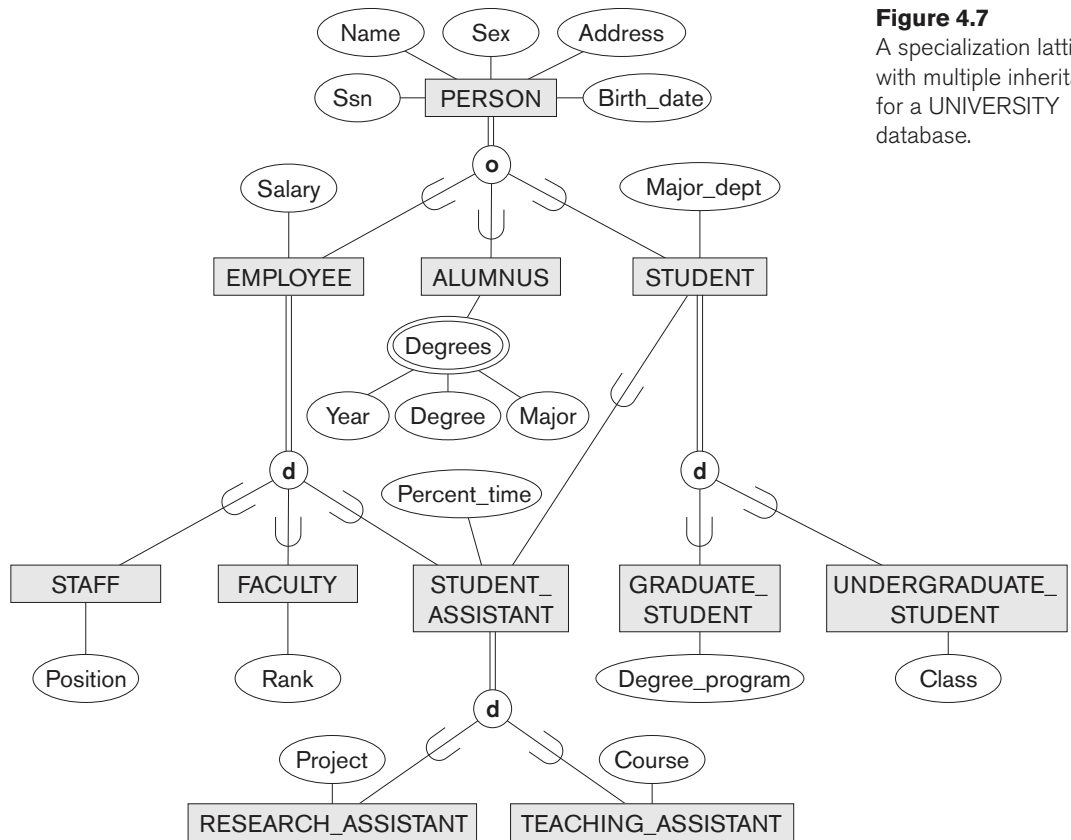
Figure 4.7
A specialization lattice with multiple inheritance for a UNIVERSITY database.

The requirements for the part of the UNIVERSITY database shown in Figure 4.7 are the following:

1. The database keeps track of three types of persons: employees, alumni, and students. A person can belong to one, two, or all three of these types. Each person has a name, SSN, sex, address, and birth date.

2. Every employee has a salary, and there are three types of employees: faculty, staff, and student assistants. Each employee belongs to exactly one of these types. For each alumnus, a record of the degree or degrees that he or she earned at the university is kept, including the name of the degree, the year granted, and the major department. Each student has a major department.

3. Each faculty has a rank, whereas each staff member has a staff position. Student assistants are classified further as either research assistants or teaching assistants, and the percent of time that they work is recorded in the database. Research assistants have their research project stored, whereas teaching assistants have the current course they work on.

4. Students are further classified as either graduate or undergraduate, with the specific attributes degree program (M.S., Ph.D., M.B.A., and so on) for graduate students and class (freshman, sophomore, and so on) for undergraduates.

In Figure 4.7, all person entities represented in the database are members of the PERSON entity type, which is specialized into the subclasses {EMPLOYEE, ALUMNUS, STUDENT}. This specialization is overlapping; for example, an alumnus may also be an employee and a student pursuing an advanced degree. The subclass STUDENT is the superclass for the specialization {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}, whereas EMPLOYEE is the superclass for the specialization {STUDENT_ASSISTANT, FACULTY, STAFF}. Notice that STUDENT_ASSISTANT is also a subclass of STUDENT. Finally, STUDENT_ASSISTANT is the superclass for the specialization into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}.

In such a specialization lattice or hierarchy, a subclass inherits the attributes not only of its direct superclass, but also of all its predecessor superclasses *all the way to the root* of the hierarchy or lattice if necessary. For example, an entity in GRADUATE_STUDENT inherits all the attributes of that entity as a STUDENT *and* as a PERSON. Notice that an entity may exist in several *leaf nodes* of the hierarchy, where a **leaf node** is a class that has *no subclasses of its own.* For example, a member of GRADUATE_STUDENT may also be a member of RESEARCH_ASSISTANT.

A subclass with *more than one* superclass is called a **shared subclass**, such as ENGINEERING_MANAGER in Figure 4.6. This leads to the concept known as **multiple inheritance**, where the shared subclass ENGINEERING_MANAGER directly inherits attributes and relationships from multiple superclasses. Notice that the existence of at least one shared subclass leads to a lattice (and hence to *multiple inheritance*); if no shared subclasses existed, we would have a hierarchy rather than a lattice and only **single inheritance** would exist. An important rule related to multiple inheritance can be illustrated by the example of the shared subclass STUDENT_ASSISTANT in Figure 4.7, which inherits attributes from both EMPLOYEE and STUDENT. Here, both EMPLOYEE and STUDENT inherit *the same attributes* from PERSON. The rule states that if an attribute (or relationship) originating in the *same superclass* (PERSON) is inherited more than once via different paths (EMPLOYEE and STUDENT) in the lattice, then it should be included only once in the shared subclass (STUDENT_ASSISTANT). Hence, the attributes of PERSON are inherited *only once* in the STUDENT_ASSISTANT subclass in Figure 4.7.

It is important to note here that some models and languages are limited to **single inheritance** and *do not allow* multiple inheritance (shared subclasses). It is also important to note that some models do not allow an entity to have multiple types, and hence an entity can be a member *of only one leaf class.*[8] In such a model, it is necessary to create additional subclasses as leaf nodes to cover all

---

[8]In some models, the class is further restricted to be a *leaf node* in the hierarchy or lattice.

possible combinations of classes that may have some entity that belongs to all these classes simultaneously. For example, in the overlapping specialization of PERSON into {EMPLOYEE, ALUMNUS, STUDENT} (or {E, A, S} for short), it would be necessary to create seven subclasses of PERSON in order to cover all possible types of entities: E, A, S, E_A, E_S, A_S, and E_A_S. Obviously, this can lead to extra complexity.

Although we have used specialization to illustrate our discussion, similar concepts *apply equally* to generalization, as we mentioned at the beginning of this section. Hence, we can also speak of **generalization hierarchies** and **generalization lattices**.

4.3.3 Utilizing Specialization and Generalization in Refining Conceptual Schemas

Now we elaborate on the differences between the specialization and generalization processes and how they are used to refine conceptual schemas during conceptual database design. In the specialization process, the database designers typically start with an entity type and then define subclasses of the entity type by successive specialization; that is, they repeatedly define more specific groupings of the entity type. For example, when designing the specialization lattice in Figure 4.7, we may first specify an entity type PERSON for a university database. Then we discover that three types of persons will be represented in the database: university employees, alumni, and students and we create the specialization {EMPLOYEE, ALUMNUS, STUDENT}. The overlapping constraint is chosen because a person may belong to more than one of the subclasses. We specialize EMPLOYEE further into {STAFF, FACULTY, STUDENT_ASSISTANT}, and specialize STUDENT into {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. Finally, we specialize STUDENT_ASSISTANT into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}. This process is called **top-down conceptual refinement**. So far, we have a hierarchy; then we realize that STUDENT_ASSISTANT is a shared subclass, since it is also a subclass of STUDENT, leading to the lattice.

It is possible to arrive at the same hierarchy or lattice from the other direction. In such a case, the process involves generalization rather than specialization and corresponds to a **bottom-up conceptual synthesis**. For example, the database designers may first discover entity types such as STAFF, FACULTY, ALUMNUS, GRADUATE_STUDENT, UNDERGRADUATE_STUDENT, RESEARCH_ASSISTANT, TEACHING_ASSISTANT, and so on; then they generalize {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT} into STUDENT; then {RESEARCH_ASSISTANT, TEACHING_ASSISTANT} into STUDENT_ASSISTANT; then {STAFF, FACULTY, STUDENT_ASSISTANT} into EMPLOYEE; and finally {EMPLOYEE, ALUMNUS, STUDENT} into PERSON.

The final design of hierarchies or lattices resulting from either process may be identical; the only difference relates to the manner or order in which the schema superclasses and subclasses were created during the design process. In practice, it is likely that a combination of the two processes is employed. Notice that the

notion of representing data and knowledge by using superclass/subclass hierarchies and lattices is quite common in knowledge-based systems and expert systems, which combine database technology with artificial intelligence techniques. For example, frame-based knowledge representation schemes closely resemble class hierarchies. Specialization is also common in software engineering design methodologies that are based on the object-oriented paradigm.

4.4 Modeling of UNION Types Using Categories

It is sometimes necessary to represent a collection of entities from different entity types. In this case, a subclass will represent a collection of entities that is a subset of the UNION of entities from distinct entity types; we call such a *subclass* a **union type** or a **category**.[9]

For example, suppose that we have three entity types: PERSON, BANK, and COMPANY. In a database for motor vehicle registration, an owner of a vehicle can be a person, a bank (holding a lien on a vehicle), or a company. We need to create a class (collection of entities) that includes entities of all three types to play the role of *vehicle owner*. A category (union type) OWNER that is a *subclass of the UNION* of the three entity sets of COMPANY, BANK, and PERSON can be created for this purpose. We display categories in an EER diagram as shown in Figure 4.8. The superclasses COMPANY, BANK, and PERSON are connected to the circle with the ∪ symbol, which stands for the *set union operation*. An arc with the subset symbol connects the circle to the (subclass) OWNER category. In Figure 4.8 we have two categories: OWNER, which is a subclass (subset) of the union of PERSON, BANK, and COMPANY; and REGISTERED_VEHICLE, which is a subclass (subset) of the union of CAR and TRUCK.

A category has two or more superclasses that may represent collections of entities from *distinct entity types*, whereas other superclass/subclass relationships always have a single superclass. To better understand the difference, we can compare a category, such as OWNER in Figure 4.8, with the ENGINEERING_MANAGER shared subclass in Figure 4.6. The latter is a subclass of *each of* the three superclasses ENGINEER, MANAGER, and SALARIED_EMPLOYEE, so an entity that is a member of ENGINEERING_MANAGER must exist in *all three collections*. This represents the constraint that an engineering manager must be an ENGINEER, a MANAGER, *and* a SALARIED_EMPLOYEE; that is, the ENGINEERING_MANAGER entity set is a subset of the *intersection* of the three entity sets. On the other hand, a category is a subset of the *union* of its superclasses. Hence, an entity that is a member of OWNER must exist in *only one* of the superclasses. This represents the constraint that an OWNER may be a COMPANY, a BANK, *or* a PERSON in Figure 4.8.

---

[9]Our use of the term *category* is based on the ECR (entity–category–relationship) model (Elmasri et al., 1985).
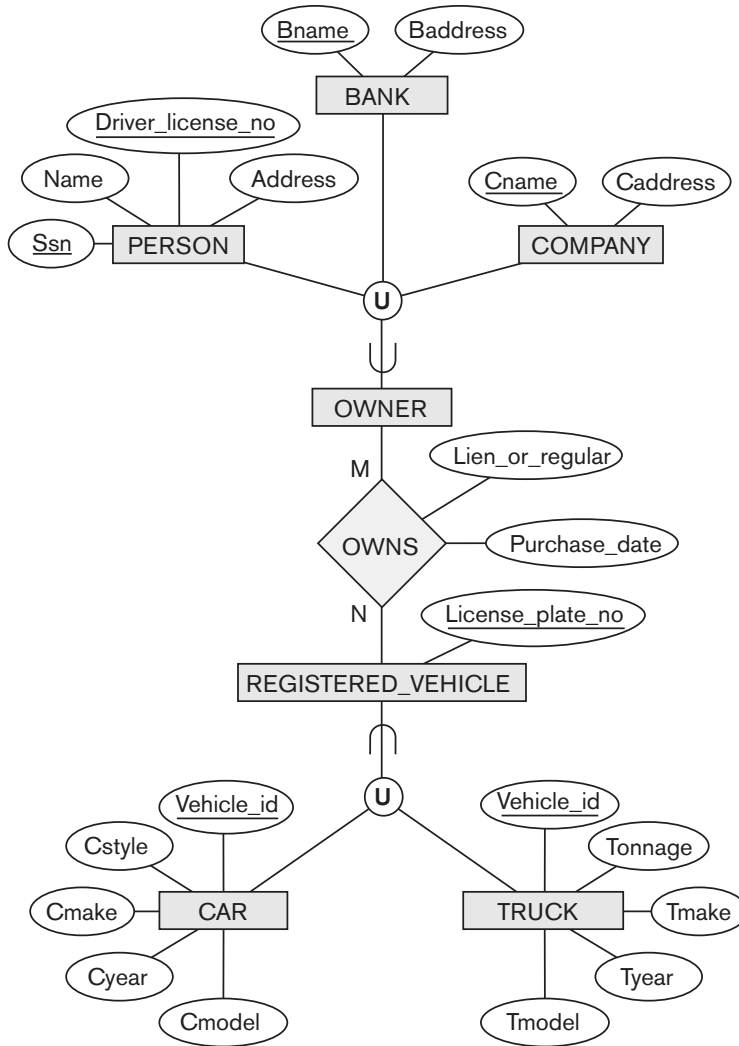
Figure 4.8
Two categories (union types): OWNER and REGISTERED_VEHICLE.

Attribute inheritance works more selectively in the case of categories. For example, in Figure 4.8 each OWNER entity inherits the attributes of a COMPANY, a PERSON, or a BANK, depending on the superclass to which the entity belongs. On the other hand, a shared subclass such as ENGINEERING_MANAGER (Figure 4.6) inherits *all* the attributes of its superclasses SALARIED_EMPLOYEE, ENGINEER, and MANAGER.

It is interesting to note the difference between the category REGISTERED_VEHICLE (Figure 4.8) and the generalized superclass VEHICLE (Figure 4.3(b)). In Figure 4.3(b), every car and every truck is a VEHICLE; but in Figure 4.8, the REGISTERED_VEHICLE category includes some cars and some trucks but not necessarily

all of them (for example, some cars or trucks may not be registered). In general, a specialization or generalization such as that in Figure 4.3(b), if it were *partial*, would not preclude VEHICLE from containing other types of entities, such as motorcycles. However, a category such as REGISTERED_VEHICLE in Figure 4.8 implies that only cars and trucks, but not other types of entities, can be members of REGISTERED_VEHICLE.

A category can be **total** or **partial**. A total category holds the *union* of all entities in its superclasses, whereas a partial category can hold a *subset of the union*. A total category is represented diagrammatically by a double line connecting the category and the circle, whereas a partial category is indicated by a single line.

The superclasses of a category may have different key attributes, as demonstrated by the OWNER category in Figure 4.8, or they may have the same key attribute, as demonstrated by the REGISTERED_VEHICLE category. Notice that if a category is total (not partial), it may be represented alternatively as a total specialization (or a total generalization). In this case, the choice of which representation to use is subjective. If the two classes represent the same type of entities and share numerous attributes, including the same key attributes, specialization/generalization is preferred; otherwise, categorization (union type) is more appropriate.

It is important to note that some modeling methodologies do not have union types. In these models, a union type must be represented in a roundabout way (see Section 9.2).

4.5 A Sample UNIVERSITY EER Schema, Design Choices, and Formal Definitions

In this section, we first give an example of a database schema in the EER model to illustrate the use of the various concepts discussed here and in Chapter 3. Then, we discuss design choices for conceptual schemas, and finally we summarize the EER model concepts and define them formally in the same manner in which we formally defined the concepts of the basic ER model in Chapter 3.

4.5.1 A Different UNIVERSITY Database Example

Consider a UNIVERSITY database that has *different requirements* from the UNIVERSITY database presented in Section 3.10. This database keeps track of students and their majors, transcripts, and registration as well as of the university's course offerings. The database also keeps track of the sponsored research projects of faculty and graduate students. This schema is shown in Figure 4.9. A discussion of the requirements that led to this schema follows.

For each person, the database maintains information on the person's Name [Name], Social Security number [Ssn], address [Address], sex [Sex], and birth date [Bdate]. Two subclasses of the PERSON entity type are identified: FACULTY and STUDENT. Specific attributes of FACULTY are rank [Rank] (assistant, associate, adjunct, research,

Figure 4.9
An EER conceptual schema for a different UNIVERSITY database.
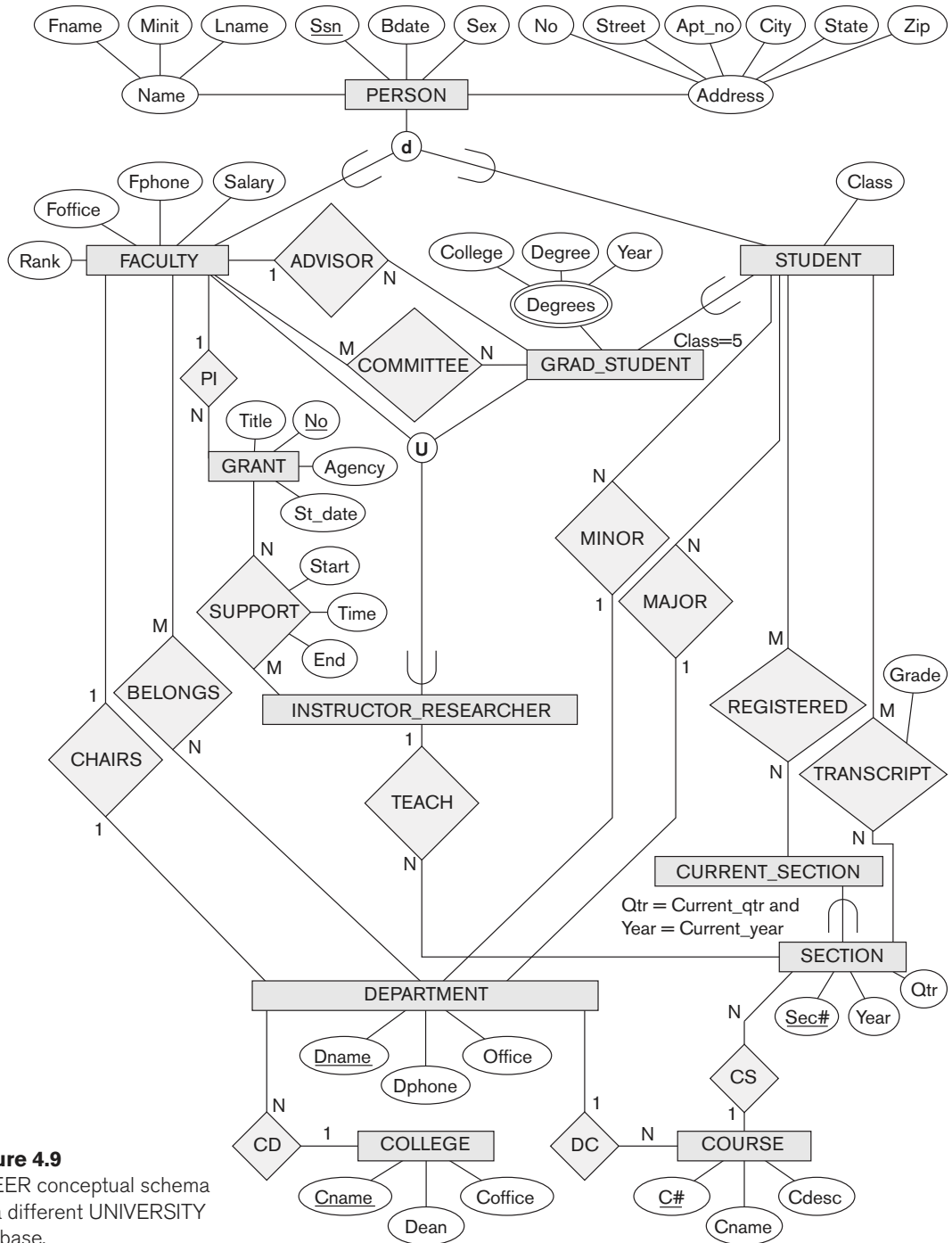
visiting, and so on), office [Foffice], office phone [Fphone], and salary [Salary]. All faculty members are related to the academic department(s) with which they are affiliated [BELONGS] (a faculty member can be associated with several departments, so the relationship is M:N). A specific attribute of STUDENT is [Class] (freshman = 1, sophomore = 2, … , MS student = 5, PhD student = 6). Each STUDENT is also related to his or her major and minor departments (if known) [MAJOR] and [MINOR], to the course sections he or she is currently attending [REGISTERED], and to the courses completed [TRANSCRIPT]. Each TRANSCRIPT instance includes the grade the student received [Grade] in a section of a course.

GRAD_STUDENT is a subclass of STUDENT, with the defining predicate (Class = 5 OR Class = 6). For each graduate student, we keep a list of previous degrees in a composite, multivalued attribute [Degrees]. We also relate the graduate student to a faculty advisor [ADVISOR] and to a thesis committee [COMMITTEE], if one exists.

An academic department has the attributes name [Dname], telephone [Dphone], and office number [Office] and is related to the faculty member who is its chairperson [CHAIRS] and to the college to which it belongs [CD]. Each college has attributes college name [Cname], office number [Coffice], and the name of its dean [Dean].

A course has attributes course number [C#], course name [Cname], and course description [Cdesc]. Several sections of each course are offered, with each section having the attributes section number [Sec#] and the year and quarter in which the section was offered ([Year] and [Qtr]).[10] Section numbers uniquely identify each section. The sections being offered during the current quarter are in a subclass CURRENT_SECTION of SECTION, with the defining predicate Qtr = Current_qtr and Year = Current_year. Each section is related to the instructor who taught or is teaching it ([TEACH]), if that instructor is in the database.

The category INSTRUCTOR_RESEARCHER is a subset of the union of FACULTY and GRAD_STUDENT and includes all faculty, as well as graduate students who are supported by teaching or research. Finally, the entity type GRANT keeps track of research grants and contracts awarded to the university. Each grant has attributes grant title [Title], grant number [No], the awarding agency [Agency], and the starting date [St_date]. A grant is related to one principal investigator [PI] and to all researchers it supports [SUPPORT]. Each instance of support has as attributes the starting date of support [Start], the ending date of the support (if known) [End], and the percentage of time being spent on the project [Time] by the researcher being supported.

4.5.2 Design Choices for Specialization/Generalization

It is not always easy to choose the most appropriate conceptual design for a database application. In Section 3.7.3, we presented some of the typical issues that confront a database designer when choosing among the concepts of entity

---

[10]We assume that the *quarter* system rather than the *semester* system is used in this university.

types, relationship types, and attributes to represent a particular miniworld situation as an ER schema. In this section, we discuss design guidelines and choices for the EER concepts of specialization/generalization and categories (union types).

As we mentioned in Section 3.7.3, conceptual database design should be considered as an iterative refinement process until the most suitable design is reached. The following guidelines can help to guide the design process for EER concepts:

- In general, many specializations and subclasses can be defined to make the conceptual model accurate. However, the drawback is that the design becomes quite cluttered. It is important to represent only those subclasses that are deemed necessary to avoid extreme cluttering of the conceptual schema.

- If a subclass has few specific (local) attributes and no specific relationships, it can be merged into the superclass. The specific attributes would hold NULL values for entities that are not members of the subclass. A *type* attribute could specify whether an entity is a member of the subclass.

- Similarly, if all the subclasses of a specialization/generalization have few specific attributes and no specific relationships, they can be merged into the superclass and replaced with one or more *type* attributes that specify the subclass or subclasses that each entity belongs to (see Section 9.2 for how this criterion applies to relational databases).

- Union types and categories should generally be avoided unless the situation definitely warrants this type of construct, which does occur in some practical situations. If possible, we try to model using specialization/generalization as discussed at the end of Section 4.4.

- The choice of disjoint/overlapping and total/partial constraints on specialization/generalization is driven by the rules in the miniworld being modeled. If the requirements do not indicate any particular constraints, the default would generally be overlapping and partial, since this does not specify any restrictions on subclass membership.

As an example of applying these guidelines, consider Figure 4.6, where no specific (local) attributes are shown. We could merge all the subclasses into the EMPLOYEE entity type and add the following attributes to EMPLOYEE:

- An attribute Job_type whose value set {'Secretary', 'Engineer', 'Technician'} would indicate which subclass in the first specialization each employee belongs to.

- An attribute Pay_method whose value set {'Salaried', 'Hourly'} would indicate which subclass in the second specialization each employee belongs to.

- ■ An attribute Is_a_manager whose value set {'Yes', 'No'} would indicate whether an individual employee entity is a manager or not.

4.5.3 Formal Definitions for the EER Model Concepts

We now summarize the EER model concepts and give formal definitions. A **class**[11] defines a type of entity and represents a set or collection of entities of that type; this includes any of the EER schema constructs that correspond to collections of entities, such as entity types, subclasses, superclasses, and categories. A **subclass** S is a class whose entities must always be a subset of the entities in another class, called the **superclass** C of the **superclass/subclass** (or **IS-A**) **relationship**. We denote such a relationship by C/S. For such a superclass/subclass relationship, we must always have

$$S \subseteq C$$

A **specialization** $Z = \{S_1, S_2, \ldots, S_n\}$ is a set of subclasses that have the same superclass G; that is, G/S_i is a superclass/subclass relationship for $i = 1, 2, \ldots, n$. G is called a **generalized entity type** (or the **superclass** of the specialization, or a **generalization** of the subclasses $\{S_1, S_2, \ldots, S_n\}$). Z is said to be **total** if we always (at any point in time) have

$$\bigcup_{i=1}^{n} S_i = G$$

Otherwise, Z is said to be **partial**. Z is said to be **disjoint** if we always have

$$S_i \cap S_j = \varnothing \text{ (empty set) for } i \neq j$$

Otherwise, Z is said to be **overlapping**.

A subclass S of C is said to be **predicate-defined** if a predicate p on the attributes of C is used to specify which entities in C are members of S; that is, $S = C[p]$, where $C[p]$ is the set of entities in C that satisfy p. A subclass that is not defined by a predicate is called **user-defined**.

A specialization Z (or generalization G) is said to be **attribute-defined** if a predicate $(A = c_i)$, where A is an attribute of G and c_i is a constant value from the domain of A, is used to specify membership in each subclass S_i in Z. Notice that if $c_i \neq c_j$ for $i \neq j$, and A is a single-valued attribute, then the specialization will be disjoint.

A **category** T is a class that is a subset of the union of n defining superclasses $D_1, D_2,$ $\ldots, D_n, n > 1$ and is formally specified as follows:

$$T \subseteq (D_1 \cup D_2 \ldots \cup D_n)$$

---

[11]The use of the word *class* here refers to a collection (set) of entities, which differs from its more common use in object-oriented programming languages such as C++. In C++, a class is a structured type definition along with its applicable functions (operations).

A predicate p_i on the attributes of D_i can be used to specify the members of each D_i that are members of T. If a predicate is specified on every D_i, we get

$$T = (D_1[p_1] \cup D_2[p_2] \, ... \cup D_n[p_n])$$

We should now extend the definition of **relationship type** given in Chapter 3 by allowing any class—not only any entity type—to participate in a relationship. Hence, we should replace the words *entity type* with *class* in that definition. The graphical notation of EER is consistent with ER because all classes are represented by rectangles.

4.6 Example of Other Notation: Representing Specialization and Generalization in UML Class Diagrams

We now discuss the UML notation for generalization/specialization and inheritance. We already presented basic UML class diagram notation and terminology in Section 3.8. Figure 4.10 illustrates a possible UML class diagram corresponding to the EER diagram in Figure 4.7. The basic notation for specialization/generalization (see Figure 4.10) is to connect the subclasses by vertical lines to a horizontal line, which has a triangle connecting the horizontal line through another vertical line to the superclass. A blank triangle indicates a specialization/generalization with the *disjoint* constraint, and a filled triangle indicates an *overlapping* constraint. The root superclass is called the **base class**, and the subclasses (leaf nodes) are called **leaf classes**.

The preceding discussion and the example in Figure 4.10, as well as the presentation in Section 3.8, gave a brief overview of UML class diagrams and terminology. We focused on the concepts that are relevant to ER and EER database modeling rather than on those concepts that are more relevant to software engineering. In UML, there are many details that we have not discussed because they are outside the scope of this text and are mainly relevant to software engineering. For example, classes can be of various types:

- Abstract classes define attributes and operations but do not have objects corresponding to those classes. These are mainly used to specify a set of attributes and operations that can be inherited.
- Concrete classes can have objects (entities) instantiated to belong to the class.
- Template classes specify a template that can be further used to define other classes.

In database design, we are mainly concerned with specifying concrete classes whose collections of objects are permanently (or persistently) stored in the database. The bibliographic notes at the end of this chapter give some references to books that describe complete details of UML.
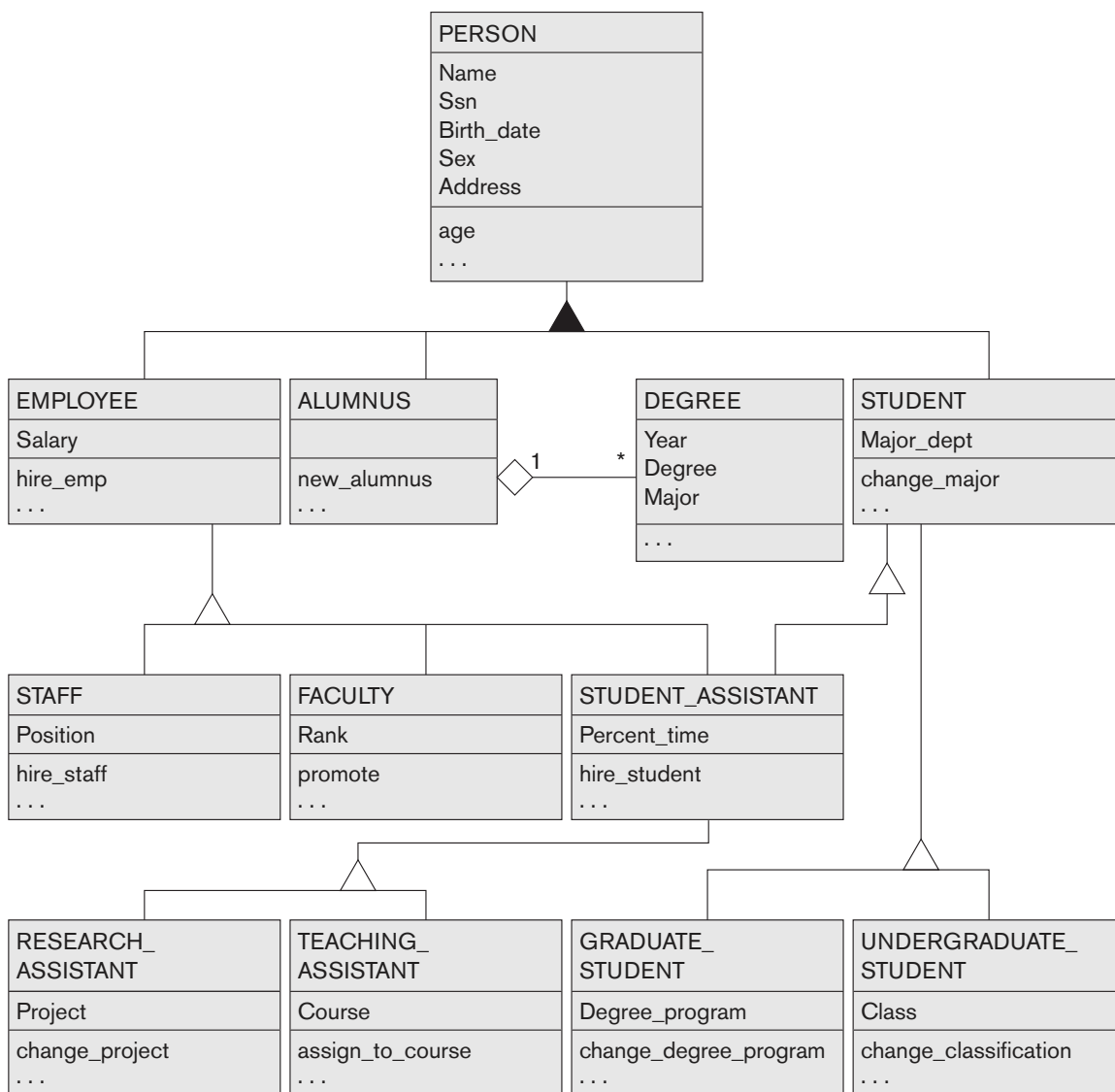
Figure 4.10
A UML class diagram corresponding to the EER diagram in Figure 4.7,
illustrating UML notation for specialization/generalization.

4.7 Data Abstraction, Knowledge Representation, and Ontology Concepts

In this section, we discuss in general terms some of the modeling concepts that we described quite specifically in our presentation of the ER and EER models in Chapter 3 and earlier in this chapter. This terminology is not only used in conceptual

data modeling but also in artificial intelligence literature when discussing **knowledge representation** (**KR**). This section discusses the similarities and differences between conceptual modeling and knowledge representation, and introduces some of the alternative terminology and a few additional concepts.

The goal of KR techniques is to develop concepts for accurately modeling some **domain of knowledge** by creating an **ontology**[12] that describes the concepts of the domain and how these concepts are interrelated. The ontology is used to store and manipulate knowledge for drawing inferences, making decisions, or answering questions. The goals of KR are similar to those of semantic data models, but there are some important similarities and differences between the two disciplines:

- Both disciplines use an abstraction process to identify common properties and important aspects of objects in the miniworld (also known as *domain of discourse* in KR) while suppressing insignificant differences and unimportant details.

- Both disciplines provide concepts, relationships, constraints, operations, and languages for defining data and representing knowledge.

- KR is generally broader in scope than semantic data models. Different forms of knowledge, such as rules (used in inference, deduction, and search), incomplete and default knowledge, and temporal and spatial knowledge, are represented in KR schemes. Database models are being expanded to include some of these concepts (see Chapter 26).

- KR schemes include **reasoning mechanisms** that deduce additional facts from the facts stored in a database. Hence, whereas most current database systems are limited to answering direct queries, knowledge-based systems using KR schemes can answer queries that involve **inferences** over the stored data. Database technology is being extended with inference mechanisms (see Section 26.5).

- Whereas most data models concentrate on the representation of database schemas, or meta-knowledge, KR schemes often mix up the schemas with the instances themselves in order to provide flexibility in representing exceptions. This often results in inefficiencies when these KR schemes are implemented, especially when compared with databases and when a large amount of structured data (facts) needs to be stored.

We now discuss four **abstraction concepts** that are used in semantic data models, such as the EER model, as well as in KR schemes: (1) classification and instantiation, (2) identification, (3) specialization and generalization, and (4) aggregation and association. The paired concepts of classification and instantiation are inverses of one another, as are generalization and specialization. The concepts of aggregation and association are also related. We discuss these abstract concepts and their relation to the concrete representations used in the EER model to clarify the data abstraction process and to improve our understanding of the related process of conceptual schema design. We close the section with a brief discussion of *ontology*, which is being used widely in recent knowledge representation research.

---

[12]An *ontology* is somewhat similar to a conceptual schema, but with more knowledge, rules, and exceptions.

4.7.1 Classification and Instantiation

The process of **classification** involves systematically assigning similar objects/entities to object classes/entity types. We can now describe (in DB) or reason about (in KR) the classes rather than the individual objects. Collections of objects that share the same types of attributes, relationships, and constraints are classified into classes in order to simplify the process of discovering their properties. **Instantiation** is the inverse of classification and refers to the generation and specific examination of distinct objects of a class. An object instance is related to its object class by the **IS-AN-INSTANCE-OF** or **IS-A-MEMBER-OF** relationship. Although EER diagrams do not display instances, the UML diagrams allow a form of instantiation by permitting the display of individual objects. We *did not* describe this feature in our introduction to UML class diagrams.

In general, the objects of a class should have a similar type structure. However, some objects may display properties that differ in some respects from the other objects of the class; these **exception objects** also need to be modeled, and KR schemes allow more varied exceptions than do database models. In addition, certain properties apply to the class as a whole and not to the individual objects; KR schemes allow such **class properties**. UML diagrams also allow specification of class properties.

In the EER model, entities are classified into entity types according to their basic attributes and relationships. Entities are further classified into subclasses and categories based on additional similarities and differences (exceptions) among them. Relationship instances are classified into relationship types. Hence, entity types, subclasses, categories, and relationship types are the different concepts that are used for classification in the EER model. The EER model does not provide explicitly for class properties, but it may be extended to do so. In UML, objects are classified into classes, and it is possible to display both class properties and individual objects.

Knowledge representation models allow multiple classification schemes in which one class is an *instance* of another class (called a **meta-class**). Notice that this *cannot* be represented directly in the EER model, because we have only two levels—classes and instances. The only relationship among classes in the EER model is a superclass/subclass relationship, whereas in some KR schemes an additional class/instance relationship can be represented directly in a class hierarchy. An instance may itself be another class, allowing multiple-level classification schemes.

4.7.2 Identification

Identification is the abstraction process whereby classes and objects are made uniquely identifiable by means of some **identifier**. For example, a class name uniquely identifies a whole class within a schema. An additional mechanism is necessary for telling distinct object instances apart by means of object identifiers. Moreover, it is necessary to identify multiple manifestations in the database of the same real-world

object. For example, we may have a tuple <'Matthew Clarke', '610618', '376-9821'> in a PERSON relation and another tuple <'301-54-0836', 'CS', 3.8> in a STUDENT relation that happen to represent the same real-world entity. There is no way to identify the fact that these two database objects (tuples) represent the same real-world entity unless we make a provision *at design time* for appropriate cross-referencing to supply this identification. Hence, identification is needed at two levels:

- To distinguish among database objects and classes
- To identify database objects and to relate them to their real-world counterparts

In the EER model, identification of schema constructs is based on a system of unique names for the constructs in a schema. For example, every class in an EER schema—whether it is an entity type, a subclass, a category, or a relationship type—must have a distinct name. The names of attributes of a particular class must also be distinct. Rules for unambiguously identifying attribute name references in a specialization or generalization lattice or hierarchy are needed as well.

At the object level, the values of key attributes are used to distinguish among entities of a particular entity type. For weak entity types, entities are identified by a combination of their own partial key values and the entities they are related to in the owner entity type(s). Relationship instances are identified by some combination of the entities that they relate to, depending on the cardinality ratio specified.

4.7.3 Specialization and Generalization

Specialization is the process of classifying a class of objects into more specialized subclasses. **Generalization** is the inverse process of generalizing several classes into a higher-level abstract class that includes the objects in all these classes. Specialization is conceptual refinement, whereas generalization is conceptual synthesis. Subclasses are used in the EER model to represent specialization and generalization. We call the relationship between a subclass and its superclass an **IS-A-SUBCLASS-OF** relationship, or simply an **IS-A** relationship. This is the same as the IS-A relationship discussed earlier in Section 4.5.3.

4.7.4 Aggregation and Association

Aggregation is an abstraction concept for building composite objects from their component objects. There are three cases where this concept can be related to the EER model. The first case is the situation in which we aggregate attribute values of an object to form the whole object. The second case is when we represent an aggregation relationship as an ordinary relationship. The third case, which the EER model does not provide for explicitly, involves the possibility of combining objects that are related by a particular relationship instance into a *higher-level aggregate object.* This is sometimes useful when the higher-level aggregate object is itself to be related to another object. We call the relationship between the primitive objects and their aggregate object **IS-A-PART-OF**; the inverse is called **IS-A-COMPONENT-OF**. UML provides for all three types of aggregation.

The abstraction of **association** is used to associate objects from several *independent classes.* Hence, it is somewhat similar to the second use of aggregation. It is represented in the EER model by relationship types, and in UML by associations. This abstract relationship is called **IS-ASSOCIATED-WITH**.

In order to understand the different uses of aggregation better, consider the ER schema shown in Figure 4.11(a), which stores information about interviews by job applicants to various companies. The class COMPANY is an aggregation of the attributes (or component objects) Cname (company name) and Caddress (company address), whereas JOB_APPLICANT is an aggregate of Ssn, Name, Address, and Phone. The relationship attributes Contact_name and Contact_phone represent the name and phone number of the person in the company who is responsible for the interview. Suppose that some interviews result in job offers, whereas others do not. We would like to treat INTERVIEW as a class to associate it with JOB_OFFER. The schema shown in Figure 4.11(b) is *incorrect* because it requires each interview relationship instance to have a job offer. The schema shown in Figure 4.11(c) is *not allowed* because the ER model does not allow relationships among relationships.

One way to represent this situation is to create a higher-level aggregate class composed of COMPANY, JOB_APPLICANT, and INTERVIEW and to relate this class to JOB_OFFER, as shown in Figure 4.11(d). Although the EER model as described in this book does not have this facility, some semantic data models do allow it and call the resulting object a **composite** or **molecular object**. Other models treat entity types and relationship types uniformly and hence permit relationships among relationships, as illustrated in Figure 4.11(c).

To represent this situation correctly in the ER model as described here, we need to create a new weak entity type INTERVIEW, as shown in Figure 4.11(e), and relate it to JOB_OFFER. Hence, we can always represent these situations correctly in the ER model by creating additional entity types, although it may be conceptually more desirable to allow direct representation of aggregation, as in Figure 4.11(d), or to allow relationships among relationships, as in Figure 4.11(c).

The main structural distinction between aggregation and association is that when an association instance is deleted, the participating objects may continue to exist. However, if we support the notion of an aggregate object—for example, a CAR that is made up of objects ENGINE, CHASSIS, and TIRES—then deleting the aggregate CAR object amounts to deleting all its component objects.

4.7.5 Ontologies and the Semantic Web

In recent years, the amount of computerized data and information available on the Web has spiraled out of control. Many different models and formats are used. In addition to the database models that we present in this text, much information is stored in the form of **documents**, which have considerably less structure than
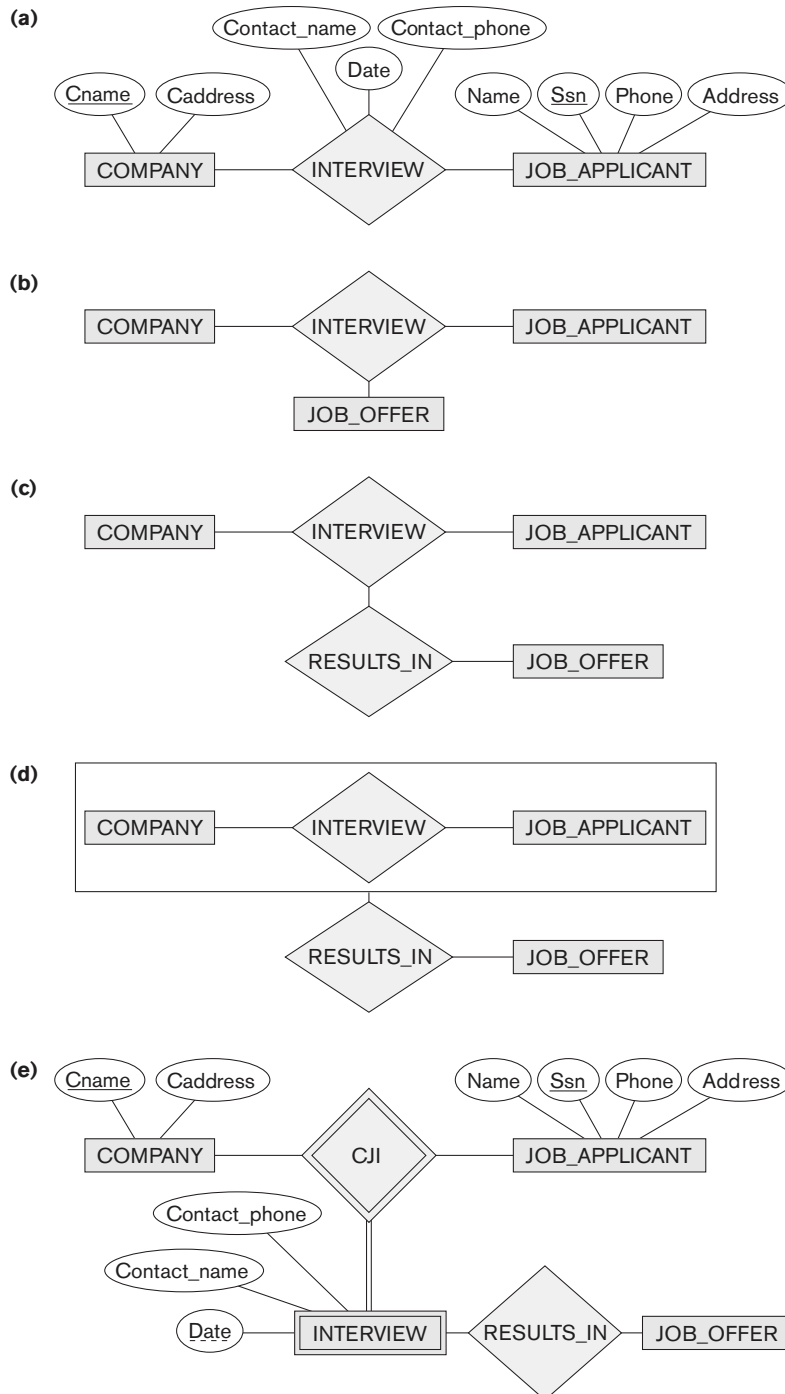
Figure 4.11
Aggregation. (a) The relationship type INTERVIEW. (b) Including JOB_OFFER in a ternary relationship type (incorrect). (c) Having the RESULTS_IN relationship participate in other relationships (not allowed in ER). (d) Using aggregation and a composite (molecular) object (generally not allowed in ER but allowed by some modeling tools). (e) Correct representation in ER.

database information does. One ongoing project that is attempting to allow information exchange among computers on the Web is called the **Semantic Web**, which attempts to create knowledge representation models that are quite general in order to allow meaningful information exchange and search among machines. The concept of *ontology* is considered to be the most promising basis for achieving the goals of the Semantic Web and is closely related to knowledge representation. In this section, we give a brief introduction to what ontology is and how it can be used as a basis to automate information understanding, search, and exchange.

The study of ontologies attempts to describe the concepts and relationships that are possible in reality through some common vocabulary; therefore, it can be considered as a way to describe the knowledge of a certain community about reality. Ontology originated in the fields of philosophy and metaphysics. One commonly used definition of **ontology** is *a specification* of a *conceptualization*.[13]

In this definition, a **conceptualization** is the set of concepts and relationships that are used to represent the part of reality or knowledge that is of interest to a community of users. **Specification** refers to the language and vocabulary terms that are used to specify the conceptualization. The ontology includes both *specification* and *conceptualization.* For example, the same conceptualization may be specified in two different languages, giving two separate ontologies. Based on this general definition, there is no consensus on what an ontology is exactly. Some possible ways to describe ontologies are as follows:

- A **thesaurus** (or even a **dictionary** or a **glossary** of terms) describes the relationships between words (vocabulary) that represent various concepts.
- A **taxonomy** describes how concepts of a particular area of knowledge are related using structures similar to those used in a specialization or generalization.
- A detailed **database schema** is considered by some to be an ontology that describes the concepts (entities and attributes) and relationships of a miniworld from reality.
- A **logical theory** uses concepts from mathematical logic to try to define concepts and their interrelationships.

Usually the concepts used to describe ontologies are similar to the concepts we discuss in conceptual modeling, such as entities, attributes, relationships, specializations, and so on. The main difference between an ontology and, say, a database schema, is that the schema is usually limited to describing a small subset of a miniworld from reality in order to store and manage data. An ontology is usually considered to be more general in that it attempts to describe a part of reality or a domain of interest (for example, medical terms, electronic-commerce applications, sports, and so on) as completely as possible.

---

[13]This definition is given in Gruber (1995).

4.8 Summary

In this chapter we discussed extensions to the ER model that improve its representational capabilities. We called the resulting model the enhanced ER or EER model. We presented the concept of a subclass and its superclass and the related mechanism of attribute/relationship inheritance. We saw how it is sometimes necessary to create additional classes of entities, either because of additional specific attributes or because of specific relationship types. We discussed two main processes for defining superclass/subclass hierarchies and lattices: specialization and generalization.

Next, we showed how to display these new constructs in an EER diagram. We also discussed the various types of constraints that may apply to specialization or generalization. The two main constraints are total/partial and disjoint/overlapping. We discussed the concept of a category or union type, which is a subset of the union of two or more classes, and we gave formal definitions of all the concepts presented.

We introduced some of the notation and terminology of UML for representing specialization and generalization. In Section 4.7, we briefly discussed the discipline of knowledge representation and how it is related to semantic data modeling. We also gave an overview and summary of the types of abstract data representation concepts: classification and instantiation, identification, specialization and generalization, and aggregation and association. We saw how EER and UML concepts are related to each of these.

Review Questions

4.1. What is a subclass? When is a subclass needed in data modeling?

4.2. Define the following terms: *superclass of a subclass*, *superclass/subclass relationship*, *IS-A relationship*, *specialization*, *generalization*, *category*, *specific (local) attributes*, and *specific relationships*.

4.3. Discuss the mechanism of attribute/relationship inheritance. Why is it useful?

4.4. Discuss user-defined and predicate-defined subclasses, and identify the differences between the two.

4.5. Discuss user-defined and attribute-defined specializations, and identify the differences between the two.

4.6. Discuss the two main types of constraints on specializations and generalizations.

4.7. What is the difference between a specialization hierarchy and a specialization lattice?

4.8. What is the difference between specialization and generalization? Why do we not display this difference in schema diagrams?

4.9. How does a category differ from a regular shared subclass? What is a category used for? Illustrate your answer with examples.

4.10. For each of the following UML terms (see Sections 3.8 and 4.6), discuss the corresponding term in the EER model, if any: *object*, *class*, *association*, *aggregation*, *generalization*, *multiplicity*, *attributes*, *discriminator*, *link*, *link attribute*, *reflexive association*, and *qualified association*.

4.11. Discuss the main differences between the notation for EER schema diagrams and UML class diagrams by comparing how common concepts are represented in each.

4.12. List the various data abstraction concepts and the corresponding modeling concepts in the EER model.

4.13. What aggregation feature is missing from the EER model? How can the EER model be further enhanced to support it?

4.14. What are the main similarities and differences between conceptual database modeling techniques and knowledge representation techniques?

4.15. Discuss the similarities and differences between an ontology and a database schema.

Exercises

4.16. Design an EER schema for a database application that you are interested in. Specify all constraints that should hold on the database. Make sure that the schema has at least five entity types, four relationship types, a weak entity type, a superclass/subclass relationship, a category, and an *n*-ary (*n* > 2) relationship type.

4.17. Consider the BANK ER schema in Figure 3.21, and suppose that it is necessary to keep track of different types of ACCOUNTS (SAVINGS_ACCTS, CHECKING_ACCTS, …) and LOANS (CAR_LOANS, HOME_LOANS, …). Suppose that it is also desirable to keep track of each ACCOUNT's TRANSACTIONS (deposits, withdrawals, checks, …) and each LOAN's PAYMENTS; both of these include the amount, date, and time. Modify the BANK schema, using ER and EER concepts of specialization and generalization. State any assumptions you make about the additional requirements.

4.18. The following narrative describes a simplified version of the organization of Olympic facilities planned for the summer Olympics. Draw an EER diagram that shows the entity types, attributes, relationships, and specializations for this application. State any assumptions you make. The Olympic facilities are divided into sports complexes. Sports complexes are divided into *one-sport* and *multisport* types. Multisport complexes have areas of the complex designated for each sport with a location indicator (e.g., center, NE corner, and so

on). A complex has a location, chief organizing individual, total occupied area, and so on. Each complex holds a series of events (e.g., the track stadium may hold many different races). For each event there is a planned date, duration, number of participants, number of officials, and so on. A roster of all officials will be maintained together with the list of events each official will be involved in. Different equipment is needed for the events (e.g., goal posts, poles, parallel bars) as well as for maintenance. The two types of facilities (one-sport and multisport) will have different types of information. For each type, the number of facilities needed is kept, together with an approximate budget.

4.19. Identify all the important concepts represented in the library database case study described below. In particular, identify the abstractions of classification (entity types and relationship types), aggregation, identification, and specialization/generalization. Specify (min, max) cardinality constraints whenever possible. List details that will affect the eventual design but that have no bearing on the conceptual design. List the semantic constraints separately. Draw an EER diagram of the library database.

Case Study: The Georgia Tech Library (GTL) has approximately 16,000 members, 100,000 titles, and 250,000 volumes (an average of 2.5 copies per book). About 10% of the volumes are out on loan at any one time. The librarians ensure that the books that members want to borrow are available when the members want to borrow them. Also, the librarians must know how many copies of each book are in the library or out on loan at any given time. A catalog of books is available online that lists books by author, title, and subject area. For each title in the library, a book description is kept in the catalog; the description ranges from one sentence to several pages. The reference librarians want to be able to access this description when members request information about a book. Library staff includes chief librarian, departmental associate librarians, reference librarians, check-out staff, and library assistants.

Books can be checked out for 21 days. Members are allowed to have only five books out at a time. Members usually return books within three to four weeks. Most members know that they have one week of grace before a notice is sent to them, so they try to return books before the grace period ends. About 5% of the members have to be sent reminders to return books. Most overdue books are returned within a month of the due date. Approximately 5% of the overdue books are either kept or never returned. The most active members of the library are defined as those who borrow books at least ten times during the year. The top 1% of membership does 15% of the borrowing, and the top 10% of the membership does 40% of the borrowing. About 20% of the members are totally inactive in that they are members who never borrow.

To become a member of the library, applicants fill out a form including their SSN, campus and home mailing addresses, and phone numbers. The librari-

ans issue a numbered, machine-readable card with the member's photo on it. This card is good for four years. A month before a card expires, a notice is sent to a member for renewal. Professors at the institute are considered automatic members. When a new faculty member joins the institute, his or her information is pulled from the employee records and a library card is mailed to his or her campus address. Professors are allowed to check out books for three-month intervals and have a two-week grace period. Renewal notices to professors are sent to their campus address.

The library does not lend some books, such as reference books, rare books, and maps. The librarians must differentiate between books that can be lent and those that cannot be lent. In addition, the librarians have a list of some books they are interested in acquiring but cannot obtain, such as rare or out-of-print books and books that were lost or destroyed but have not been replaced. The librarians must have a system that keeps track of books that cannot be lent as well as books that they are interested in acquiring. Some books may have the same title; therefore, the title cannot be used as a means of identification. Every book is identified by its International Standard Book Number (ISBN), a unique international code assigned to all books. Two books with the same title can have different ISBNs if they are in different languages or have different bindings (hardcover or softcover). Editions of the same book have different ISBNs.

The proposed database system must be designed to keep track of the members, the books, the catalog, and the borrowing activity.

4.20. Design a database to keep track of information for an art museum. Assume that the following requirements were collected:

- The museum has a collection of ART_OBJECTS. Each ART_OBJECT has a unique Id_no, an Artist (if known), a Year (when it was created, if known), a Title, and a Description. The art objects are categorized in several ways, as discussed below.

- ART_OBJECTS are categorized based on their type. There are three main types—PAINTING, SCULPTURE, and STATUE—plus another type called OTHER to accommodate objects that do not fall into one of the three main types.

- A PAINTING has a Paint_type (oil, watercolor, etc.), material on which it is Drawn_on (paper, canvas, wood, etc.), and Style (modern, abstract, etc.).

- A SCULPTURE or a statue has a Material from which it was created (wood, stone, etc.), Height, Weight, and Style.

- An art object in the OTHER category has a Type (print, photo, etc.) and Style.

- ART_OBJECTs are categorized as either PERMANENT_COLLECTION (objects that are owned by the museum) and BORROWED. Information captured about objects in the PERMANENT_COLLECTION includes Date_acquired, Status (on display, on loan, or stored), and Cost. Information

captured about BORROWED objects includes the Collection from which it was borrowed, Date_borrowed, and Date_returned.

■ Information describing the country or culture of Origin (Italian, Egyptian, American, Indian, and so forth) and Epoch (Renaissance, Modern, Ancient, and so forth) is captured for each ART_OBJECT.

■ The museum keeps track of ARTIST information, if known: Name, DateBorn (if known), Date_died (if not living), Country_of_origin, Epoch, Main_style, and Description. The Name is assumed to be unique.

■ Different EXHIBITIONS occur, each having a Name, Start_date, and End_date. EXHIBITIONS are related to all the art objects that were on display during the exhibition.

■ Information is kept on other COLLECTIONS with which the museum interacts; this information includes Name (unique), Type (museum, personal, etc.), Description, Address, Phone, and current Contact_person.

Draw an EER schema diagram for this application. Discuss any assumptions you make, and then justify your EER design choices.

4.21. Figure 4.12 shows an example of an EER diagram for a small-private-airport database; the database is used to keep track of airplanes, their owners, airport employees, and pilots. From the requirements for this database, the following information was collected: Each AIRPLANE has a registration number [Reg#], is of a particular plane type [OF_TYPE], and is stored in a particular hangar [STORED_IN]. Each PLANE_TYPE has a model number [Model], a capacity [Capacity], and a weight [Weight]. Each HANGAR has a number [Number], a capacity [Capacity], and a location [Location]. The database also keeps track of the OWNERs of each plane [OWNS] and the EMPLOYEEs who have maintained the plane [MAINTAIN]. Each relationship instance in OWNS relates an AIRPLANE to an OWNER and includes the purchase date [Pdate]. Each relationship instance in MAINTAIN relates an EMPLOYEE to a service record [SERVICE]. Each plane undergoes service many times; hence, it is related by [PLANE_SERVICE] to a number of SERVICE records. A SERVICE record includes as attributes the date of maintenance [Date], the number of hours spent on the work [Hours], and the type of work done [Work_code]. We use a weak entity type [SERVICE] to represent airplane service, because the airplane registration number is used to identify a service record. An OWNER is either a person or a corporation. Hence, we use a union type (category) [OWNER] that is a subset of the union of corporation [CORPORATION] and person [PERSON] entity types. Both pilots [PILOT] and employees [EMPLOYEE] are subclasses of PERSON. Each PILOT has specific attributes license number [Lic_num] and restrictions [Restr]; each EMPLOYEE has specific attributes salary [Salary] and shift worked [Shift]. All PERSON entities in the database have data kept on their Social Security number [Ssn], name [Name], address [Address], and telephone number [Phone]. For CORPORATION entities, the data kept includes name [Name], address [Address], and telephone number [Phone]. The database also keeps track of the types of
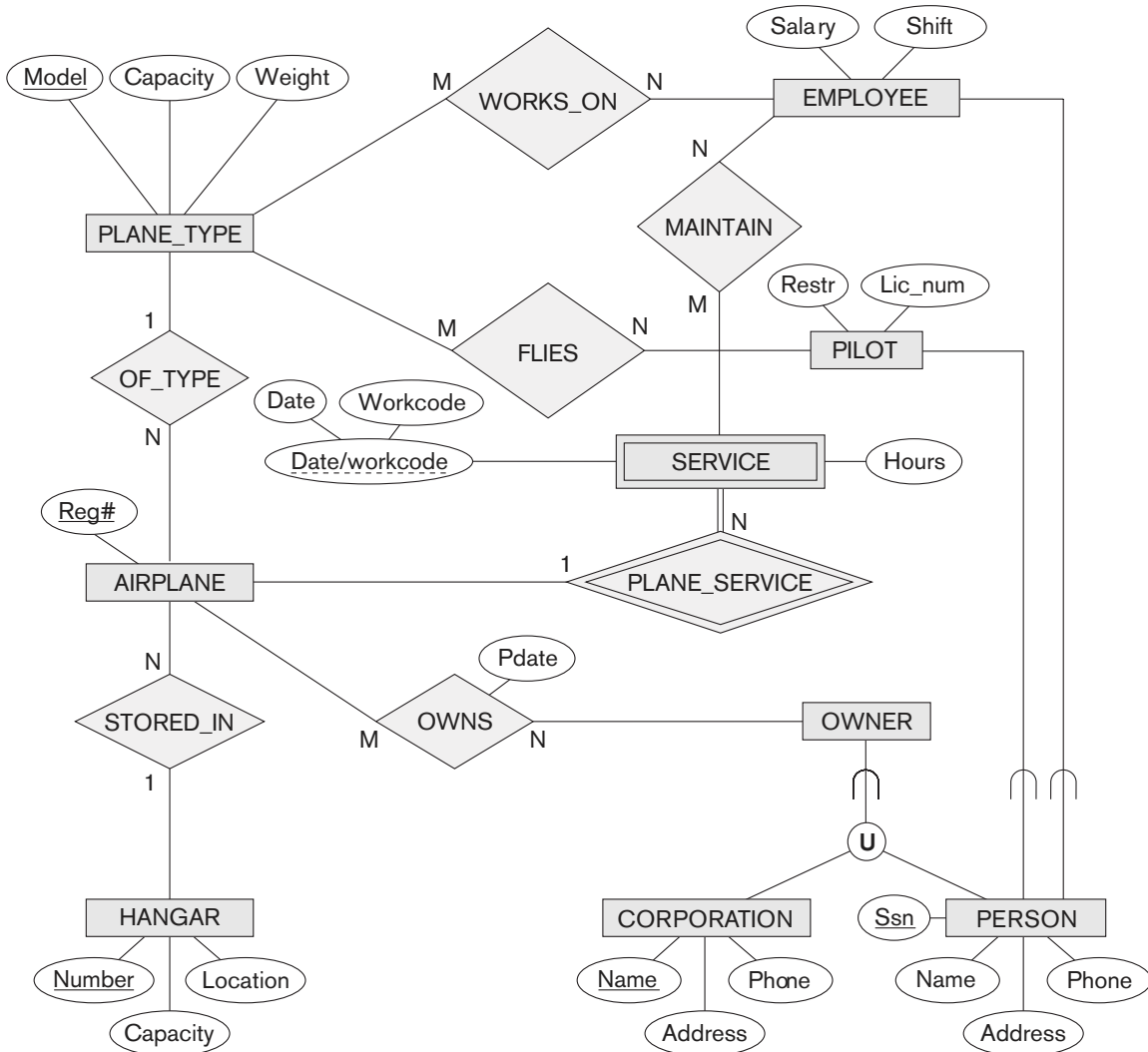
Figure 4.12
EER schema for a SMALL_AIRPORT database.

planes each pilot is authorized to fly [FLIES] and the types of planes each employee can do maintenance work on [WORKS_ON]. Show how the SMALL_AIRPORT EER schema in Figure 4.12 may be represented in UML notation. (*Note*: We have not discussed how to represent categories (union types) in UML, so you do not have to map the categories in this and the following question.)

4.22. Show how the UNIVERSITY EER schema in Figure 4.9 may be represented in UML notation.

4.23. Consider the entity sets and attributes shown in the following table. Place a checkmark in one column in each row to indicate the relationship between the far left and far right columns.

a. The left side has a relationship with the right side.

b. The right side is an attribute of the left side.

c. The left side is a specialization of the right side.

d. The left side is a generalization of the right side.

Entity Set	(a) Has a Relationship with	(b) Has an Attribute that is	(c) Is a Specialization of	(d) Is a Generalization of	Entity Set or Attribute
1. MOTHER					PERSON
2. DAUGHTER					MOTHER
3. STUDENT					PERSON
4. STUDENT					Student_id
5. SCHOOL					STUDENT
6. SCHOOL					CLASS_ROOM
7. ANIMAL					HORSE
8. HORSE					Breed
9. HORSE					Age
10. EMPLOYEE					SSN
11. FURNITURE					CHAIR
12. CHAIR					Weight
13. HUMAN					WOMAN
14. SOLDIER					PERSON
15. ENEMY_COMBATANT					PERSON

4.24. Draw a UML diagram for storing a played game of chess in a database. You may look at http://www.chessgames.com for an application similar to what you are designing. State clearly any assumptions you make in your UML diagram. A sample of assumptions you can make about the scope is as follows:

1. The game of chess is played between two players.

2. The game is played on an 8 × 8 board like the one shown below:
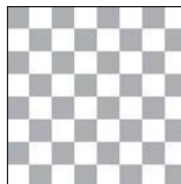
3. The players are assigned a color of black or white at the start of the game.

4. Each player starts with the following pieces (traditionally called chessmen):

   a. king
   b. queen
   c. 2 rooks
   d. 2 bishops
   e. 2 knights
   f. 8 pawns

5. Every piece has its own initial position.

6. Every piece has its own set of legal moves based on the state of the game. You do not need to worry about which moves are or are not legal except for the following issues:

   a. A piece may move to an empty square or capture an opposing piece.
   b. If a piece is captured, it is removed from the board.
   c. If a pawn moves to the last row, it is "promoted" by converting it to another piece (queen, rook, bishop, or knight).

   Note: Some of these functions may be spread over multiple classes.

4.25. Draw an EER diagram for a game of chess as described in Exercise 4. 24. Focus on persistent storage aspects of the system. For example, the system would need to retrieve all the moves of every game played in sequential order.

4.26. Which of the following EER diagrams is/are incorrect and why? State clearly any assumptions you make.

a.
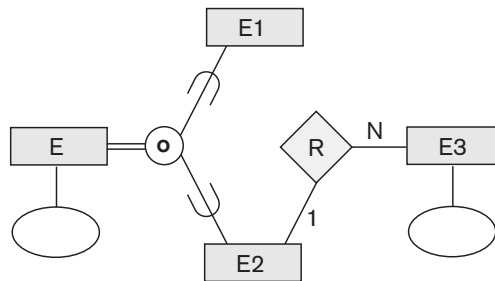
b.
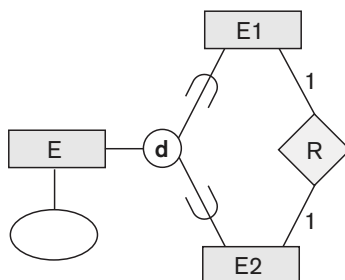
c.
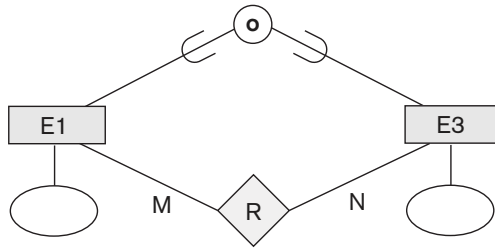
4.27. Consider the following EER diagram that describes the computer systems at a company. Provide your own attributes and key for each entity type. Supply max cardinality constraints justifying your choice. Write a complete narrative description of what this EER diagram represents.
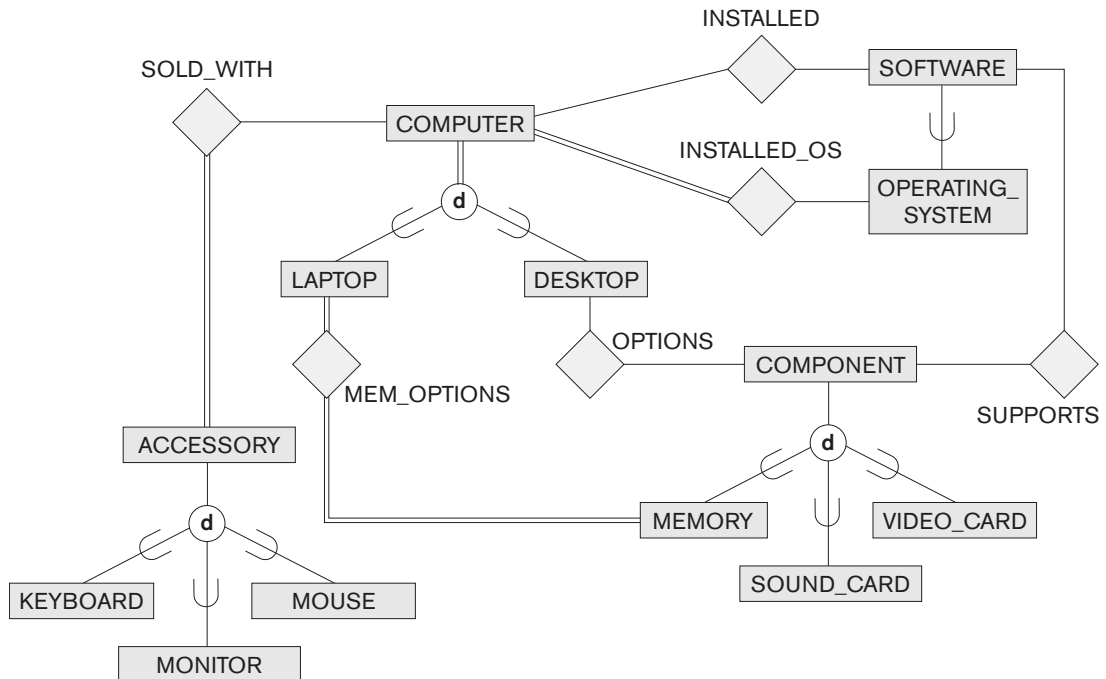
Laboratory Exercises

4.28. Consider a GRADE_BOOK database in which instructors within an academic department record points earned by individual students in their classes. The data requirements are summarized as follows:

- Each student is identified by a unique identifier, first and last name, and an e-mail address.
- Each instructor teaches certain courses each term. Each course is identified by a course number, a section number, and the term in which it is taught. For

each course he or she teaches, the instructor specifies the minimum number of points required in order to earn letter grades A, B, C, D, and F. For example, 90 points for an A, 80 points for a B, 70 points for a C, and so forth.

- Students are enrolled in each course taught by the instructor.

- Each course has a number of grading components (such as midterm exam, final exam, project, and so forth). Each grading component has a maximum number of points (such as 100 or 50) and a weight (such as 20% or 10%). The weights of all the grading components of a course usually total 100.

- Finally, the instructor records the points earned by each student in each of the grading components in each of the courses. For example, student 1234 earns 84 points for the midterm exam grading component of the section 2 course CSc2310 in the fall term of 2009. The midterm exam grading component may have been defined to have a maximum of 100 points and a weight of 20% of the course grade.

Design an enhanced entity–relationship diagram for the grade book database and build the design using a data modeling tool such as ERwin or Rational Rose.

4.29. Consider an ONLINE_AUCTION database system in which members (buyers and sellers) participate in the sale of items. The data requirements for this system are summarized as follows:

- The online site has members, each of whom is identified by a unique member number and is described by an e-mail address, name, password, home address, and phone number.

- A member may be a buyer or a seller. A buyer has a shipping address recorded in the database. A seller has a bank account number and routing number recorded in the database.

- Items are placed by a seller for sale and are identified by a unique item number assigned by the system. Items are also described by an item title, a description, starting bid price, bidding increment, the start date of the auction, and the end date of the auction.

- Items are also categorized based on a fixed classification hierarchy (for example, a modem may be classified as COMPUTER → HARDWARE → MODEM).

- Buyers make bids for items they are interested in. Bid price and time of bid are recorded. The bidder at the end of the auction with the highest bid price is declared the winner, and a transaction between buyer and seller may then proceed.

- The buyer and seller may record feedback regarding their completed transactions. Feedback contains a rating of the other party participating in the transaction (1–10) and a comment.

Design an enhanced entity–relationship diagram for the ONLINE_AUCTION database and build the design using a data modeling tool such as ERwin or Rational Rose.

4.30. Consider a database system for a baseball organization such as the major leagues. The data requirements are summarized as follows:

- The personnel involved in the league include players, coaches, managers, and umpires. Each is identified by a unique personnel id. They are also described by their first and last names along with the date and place of birth.

- Players are further described by other attributes such as their batting orientation (left, right, or switch) and have a lifetime batting average (BA).

- Within the players group is a subset of players called pitchers. Pitchers have a lifetime ERA (earned run average) associated with them.

- Teams are uniquely identified by their names. Teams are also described by the city in which they are located and the division and league in which they play (such as Central division of the American League).

- Teams have one manager, a number of coaches, and a number of players.

- Games are played between two teams, with one designated as the home team and the other the visiting team on a particular date. The score (runs, hits, and errors) is recorded for each team. The team with the most runs is declared the winner of the game.

- With each finished game, a winning pitcher and a losing pitcher are recorded. In case there is a save awarded, the save pitcher is also recorded.

- With each finished game, the number of hits (singles, doubles, triples, and home runs) obtained by each player is also recorded.

Design an enhanced entity–relationship diagram for the BASEBALL database and enter the design using a data modeling tool such as ERwin or Rational Rose.

4.31. Consider the EER diagram for the UNIVERSITY database shown in Figure 4.9. Enter this design using a data modeling tool such as ERwin or Rational Rose. Make a list of the differences in notation between the diagram in the text and the corresponding equivalent diagrammatic notation you end up using with the tool.

4.32. Consider the EER diagram for the small AIRPORT database shown in Figure 4.12. Build this design using a data modeling tool such as ERwin or Rational Rose. Be careful how you model the category OWNER in this diagram. (*Hint*: Consider using CORPORATION_IS_OWNER and PERSON_IS_ OWNER as two distinct relationship types.)

4.33. Consider the UNIVERSITY database described in Exercise 3.16. You already developed an ER schema for this database using a data modeling tool such as

ERwin or Rational Rose in Lab Exercise 3.31. Modify this diagram by classifying COURSES as either UNDERGRAD_COURSES or GRAD_COURSES and INSTRUCTORS as either JUNIOR_PROFESSORS or SENIOR_PROFESSORS. Include appropriate attributes for these new entity types. Then establish relationships indicating that junior instructors teach undergraduate courses whereas senior instructors teach graduate courses.

Selected Bibliography

Many papers have proposed conceptual or semantic data models. We give a representative list here. One group of papers, including Abrial (1974), Senko's DIAM model (1975), the NIAM method (Verheijen and VanBekkum 1982), and Bracchi et al. (1976), presents semantic models that are based on the concept of binary relationships. Another group of early papers discusses methods for extending the relational model to enhance its modeling capabilities. This includes the papers by Schmid and Swenson (1975), Navathe and Schkolnick (1978), Codd's RM/T model (1979), Furtado (1978), and the structural model of Wiederhold and Elmasri (1979).

The ER model was proposed originally by Chen (1976) and is formalized in Ng (1981). Since then, numerous extensions of its modeling capabilities have been proposed, as in Scheuermann et al. (1979), Dos Santos et al. (1979), Teorey et al. (1986), Gogolla and Hohenstein (1991), and the entity–category–relationship (ECR) model of Elmasri et al. (1985). Smith and Smith (1977) present the concepts of generalization and aggregation. The semantic data model of Hammer and McLeod (1981) introduces the concepts of class/subclass lattices, as well as other advanced modeling concepts.

A survey of semantic data modeling appears in Hull and King (1987). Eick (1991) discusses design and transformations of conceptual schemas. Analysis of constraints for n-ary relationships is given in Soutou (1998). UML is described in detail in Booch, Rumbaugh, and Jacobson (1999). Fowler and Scott (2000) and Stevens and Pooley (2000) give concise introductions to UML concepts.

Fensel (2000, 2003) discusses the Semantic Web and application of ontologies. Uschold and Gruninger (1996) and Gruber (1995) discuss ontologies. The June 2002 issue of *Communications of the ACM* is devoted to ontology concepts and applications. Fensel (2003) discusses ontologies and e-commerce.

part **3**

The Relational Data Model and SQL

This page intentionally left blank

The Relational Data Model and Relational Database Constraints

This chapter opens Part 3 of the book, which covers relational databases. The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd, 1970), and it attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a *mathematical relation*—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic. In this chapter we discuss the basic characteristics of the model and its constraints.

The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems, as well as a number of open source systems. Current popular commercial relational DBMSs (RDBMSs) include DB2 (from IBM), Oracle (from Oracle), Sybase DBMS (now from SAP), and SQLServer and Microsoft Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

Because of the importance of the relational model, all of Part 2 is devoted to this model and some of the languages associated with it. In Chapters 6 and 7, we describe some aspects of SQL, which is a comprehensive model and language that is the *standard* for commercial relational DBMSs. (Additional aspects of SQL will be covered in other chapters.) Chapter 8 covers the operations of the relational algebra and introduces the relational calculus—these are two formal languages associated with the relational model. The relational calculus is considered to be the basis for the SQL language, and the relational algebra is used in the internals of many database implementations for query processing and optimization (see Part 8 of the book).

Other features of the relational model are presented in subsequent parts of the book. Chapter 9 relates the relational model data structures to the constructs of the ER and EER models (presented in Chapters 3 and 4), and presents algorithms for designing a relational database schema by mapping a conceptual schema in the ER or EER model into a relational representation. These mappings are incorporated into many database design and CASE[1] tools. Chapters 10 and 11 in Part 4 discuss the programming techniques used to access database systems and the notion of connecting to relational databases via ODBC and JDBC standard protocols. We also introduce the topic of Web database programming in Chapter 11. Chapters 14 and 15 in Part 6 present another aspect of the relational model, namely the formal constraints of functional and multivalued dependencies; these dependencies are used to develop a relational database design theory based on the concept known as *normalization.*

In this chapter, we concentrate on describing the basic principles of the relational model of data. We begin by defining the modeling concepts and notation of the relational model in Section 5.1. Section 5.2 is devoted to a discussion of relational constraints that are considered an important part of the relational model and are automatically enforced in most relational DBMSs. Section 5.3 defines the update operations of the relational model, discusses how violations of integrity constraints are handled, and introduces the concept of a transaction. Section 5.4 summarizes the chapter.

This chapter and Chapter 8 focus on the formal foundations of the relational model, whereas Chapters 6 and 7 focus on the SQL practical relational model, which is the basis of most commercial and open source relational DBMSs. Many concepts are common between the formal and practical models, but a few differences exist that we shall point out.

5.1 Relational Model Concepts

The relational model represents the database as a collection of *relations.* Informally, each relation resembles a table of values or, to some extent, a *flat* file of records. It is called a **flat file** because each record has a simple linear or *flat* structure. For example, the database of files that was shown in Figure 1.2 is similar to the basic relational model representation. However, there are important differences between relations and files, as we shall soon see.

When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row. For example, the first table of Figure 1.2 is called STUDENT because each row represents facts about a particular student entity. The column names—Name, Student_number,

---

[1]CASE stands for computer-aided software engineering.

Class, and Major—specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a *tuple,* a column header is called an *attribute,* and the table is called a *relation.* The data type describing the types of values that can appear in each column is represented by a *domain* of possible values. We now define these terms—*domain, tuple, attribute,* and *relation*—formally.

5.1.1 Domains, Attributes, Tuples, and Relations

A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:

- Usa_phone_numbers. The set of ten-digit phone numbers valid in the United States.
- Local_phone_numbers. The set of seven-digit phone numbers valid within a particular area code in the United States. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.
- Social_security_numbers. The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- Names: The set of character strings that represent names of persons.
- Grade_point_averages. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.
- Employee_ages. Possible ages of employees in a company; each must be an integer value between 15 and 80.
- Academic_department_names. The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- Academic_department_codes. The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'.

The preceding are called *logical* definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain Usa_phone_numbers can be declared as a character string of the form (*ddd*)*ddd-dddd*, where each *d* is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80. For Academic_department_names, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as Person_weights should have the units of measurement, such as pounds or kilograms.

A **relation schema**[2] R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes, $A_1, A_2, \ldots, A_n$. Each **attribute** A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by **dom**(A_i). A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

> STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

Using the data type of each attribute, the definition is sometimes written as:

> STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. In the preceding definition, we showed assignment of generic types such as string or integer to the attributes. More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation: dom(Name) = Names; dom(Ssn) = Social_security_numbers; dom(HomePhone) = USA_phone_numbers[3], dom(Office_phone) = USA_phone_numbers, and dom(Gpa) = Grade_point_averages. It is also possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is Ssn, whereas the fourth attribute is Address.

A **relation** (or **relation state**)[4] r of the relation schema $R(A_1, A_2, \ldots, A_n)$, also denoted by $r(R)$, is a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$. Each n-**tuple** t is an ordered list of n values $t = <v_1, v_2, \ldots, v_n>$, where each value v_i, $1 \leq i \leq n$, is an element of dom (A_i) or is a special NULL value. (NULL values are discussed further below and in Section 5.1.2.) The ith value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

Figure 5.1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column. *NULL values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.

---

[2]A relation schema is sometimes called a **relation scheme**.

[3]With the large increase in phone numbers caused by the proliferation of mobile phones, most metropolitan areas in the United States now have multiple area codes, so seven-digit local dialing has been discontinued in most areas. We changed this domain to Usa_phone_numbers instead of Local_phone_numbers, which would be a more general choice. This illustrates how database requirements can change over time.

[4]This has also been called a **relation instance**. We will not use this term because *instance* is also used to refer to a single tuple or row.
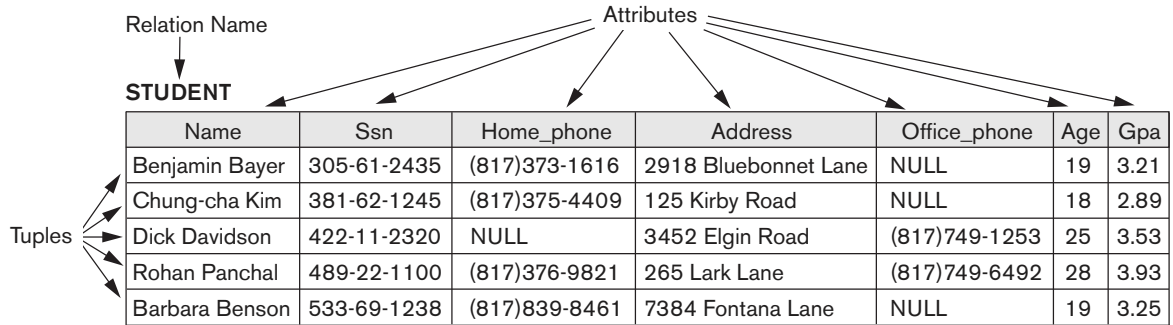
Figure 5.1
The attributes and tuples of a relation STUDENT.

The earlier definition of a relation can be *restated* more formally using set theory concepts as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains $dom(A_1)$, $dom(A_2)$, ... , $dom(A_n)$, which is a **subset** of the **Cartesian product** (denoted by $\times$) of the domains that define R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times ... \times (dom(A_n))$$

The Cartesian product specifies all possible combinations of values from the under-lying domains. Hence, if we denote the total number of values, or **cardinality,** in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|dom(A_1)| \times |dom(A_2)| \times ... \times |dom(A_n)|$$

This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. Of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real world changes, so does the relation state, by being transformed into another relation state. However, the schema R is relatively static and changes *very* infrequently—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

It is possible for several attributes to *have the same domain.* The attribute names indicate different **roles**, or interpretations, for the domain. For example, in the STUDENT relation, the same domain USA_phone_numbers plays the role of Home_phone, referring to the *home phone of a student*, and the role of Office_phone, referring to the *office phone of the student*. A third possible attribute (not shown) with the same domain could be Mobile_phone.

5.1.2 Characteristics of Relations

The earlier definition of relations implies certain characteristics that make a rela-tion different from a file or a table. We now discuss some of these characteristics.

Ordering of Tuples in a Relation. A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 5.1 could be ordered by values of Name, Ssn, Age, or some other attribute. The definition of a relation does not specify any order: There is *no preference* for one ordering over another. Hence, the relation displayed in Figure 5.2 is considered *identical* to the one shown in Figure 5.1. When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

Ordering of Values within a Tuple and an Alternative Definition of a Relation. According to the preceding definition of a relation, an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary*. In this definition, a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a *set* of attributes (instead of an ordered list of attributes), and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \ldots, t_m\}$, where each tuple t_i is a **mapping** from R to D, and D is the **union** (denoted by $\cup$) of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \ldots \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r. Each mapping t_i is called a tuple.

According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is *not* important, because the *attribute name* appears with its *value*. By this definition, the

Figure 5.2
The relation STUDENT from Figure 5.1 with a different order of tuples.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

$t = <$ (Name, Dick Davidson),(Ssn, 422-11-2320),(Home_phone, NULL),(Address, 3452 Elgin Road),
(Office_phone, (817)749-1253),(Age, 25),(Gpa, 3.53)$>$

$t = <$ (Address, 3452 Elgin Road),(Name, Dick Davidson),(Ssn, 422-11-2320),(Age, 25),
(Office_phone, (817)749-1253),(Gpa, 3.53),(Home_phone, NULL)$>$

Figure 5.3
Two identical tuples when the order of attributes and values is not part of relation definition.

two tuples shown in Figure 5.3 are identical. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appear before another in a tuple. When the attribute name and value are included together in a tuple, it is known as **self-describing data**, because the description of each value (attribute name) is included in the tuple.

We will mostly use the **first definition** of relation, where the attributes are *ordered* in the relation schema and the values within tuples *are similarly ordered,* because it simplifies much of the notation. However, the alternative definition given here is more general.[5]

Values and NULLs in the Tuples. Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes (see Chapter 3) are not allowed. This model is sometimes called the **flat relational model**. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.[6] Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.[7]

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, in Figure 5.1, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*). In general, we can have several meanings for NULL values, such as *value unknown,* **value** exists but is *not available*, or *attribute does not apply* to this tuple *(*also known as *value undefined).* An example of the last type of NULL will occur if we add an attribute Visa_status to the STUDENT relation that applies only to tuples representing foreign students. It is possible to devise different codes for different meanings of

---

[5]We will use the alternative definition of relation when we discuss query processing and optimization in Chapter 18.

[6]We discuss this assumption in more detail in Chapter 14.

[7]Extensions of the relational model remove these restrictions. For example, object-relational systems (Chapter 12) allow complex-structured attributes, as do the **non-first normal form** or **nested** relational models.

NULL values. Incorporating different types of NULL values into relational model operations has proven difficult and is outside the scope of our presentation.

The exact meaning of a NULL value governs how it fares during arithmetic aggregations or comparisons with other values. For example, a comparison of two NULL values leads to ambiguities—if both Customer A and B have NULL addresses, it *does not mean* they have the same address. During database design, it is best to avoid NULL values as much as possible. We will discuss this further in Chapters 7 and 8 in the context of operations and queries, and in Chapter 14 in the context of database design and normalization.

Interpretation (Meaning) of a Relation. The relation schema can be interpreted as a declaration or a type of **assertion**. For example, the schema of the STUDENT relation of Figure 5.1 asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. For example, the first tuple in Figure 5.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.* For example, a relation schema MAJORS (Student_ssn, Department_code) asserts that students major in academic disciplines. A tuple in this relation relates a student to his or her major discipline. Hence, the relational model represents facts about both entities and relationships *uniformly* as relations. This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type. We introduced the entity–relationship (ER) model in detail in Chapter 3, where the entity and relationship concepts were described in detail. The mapping procedures in Chapter 9 show how different constructs of the ER/EER conceptual data models (see Part 2) get converted to relations.

An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. For example, the predicate STUDENT (Name, Ssn, …) is true for the five tuples in relation STUDENT of Figure 5.1. These tuples represent five different propositions or facts in the real world. This interpretation is quite useful in the context of logical programming languages, such as Prolog, because it allows the relational model to be used within these languages (see Section 26.5). An assumption called **the closed world assumption** states that the only true facts in the universe are those present within the extension (state) of the relation(s). Any other combination of values makes the predicate false. This interpretation is useful when we consider queries on relations based on relational calculus in Section 8.6.

5.1.3 Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, … , A_n)$.

- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation—the *current relation state*—whereas STUDENT(Name, Ssn, …) refers *only* to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation $R.A$—for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.
- An n-tuple t in a relation $r(R)$ is denoted by $t = <v_1, v_2, … , v_n>$, where v_i is the value corresponding to attribute A_i. The following notation refers to **component values** of tuples:
    - Both $t[A_i]$ and $t.A_i$ (and sometimes $t[i]$) refer to the value v_i in t for attribute A_i.
    - Both $t[A_u, A_w, … , A_z]$ and $t.(A_u, A_w, … , A_z)$, where $A_u, A_w, … , A_z$ is a list of attributes from R, refer to the subtuple of values $<v_u, v_w, … , v_z>$ from t corresponding to the attributes specified in the list.

As an example, consider the tuple $t = <$'Barbara Benson', '533-69-1238', '(817)839-8461', '7384 Fontana Lane', NULL, 19, 3.25$>$ from the STUDENT relation in Figure 5.1; we have t[Name] $= <$'Barbara Benson'$>$, and t[Ssn, Gpa, Age] $= <$'533-69-1238', 3.25, 19$>$.

5.2 Relational Model Constraints and Relational Database Schemas

So far, we have discussed the characteristics of single relations. In a relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents, as we discussed in Section 1.6.8.

In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

1. Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
2. Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL (data definition language, see Section 2.3.1). We call these **schema-based constraints** or **explicit constraints**.

3. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

The characteristics of relations that we discussed in Section 5.1.2 are the inherent constraints of the relational model and belong to the first category. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint. The constraints we discuss in this section are of the second category, namely, constraints that can be expressed in the schema of the relational model via the DDL. Constraints in the third category are more general, relate to the meaning as well as behavior of attributes, and are difficult to express and enforce within the data model, so they are usually checked within the application programs that perform database updates. In some cases, these constraints can be specified as **assertions** in SQL (see Chapter 7).

Another important category of constraints is *data dependencies*, which include *functional dependencies* and *multivalued dependencies*. They are used mainly for testing the "goodness" of the design of a relational database and are utilized in a process called *normalization*, which is discussed in Chapters 14 and 15.

The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

5.2.1 Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain dom(A). We have already discussed the ways in which domains can be specified in Section 5.1.1. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double-precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and other special data types. Domains can also be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed. Rather than describe these in detail here, we discuss the data types offered by the SQL relational standard in Section 6.1.

5.2.2 Key Constraints and Constraints on NULL Values

In the formal relational model, a *relation* is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples t_1 and t_2 in a relation state r of R, we have the constraint that:

$$t_1[\text{SK}] \neq t_2[\text{SK}]$$

Any such set of attributes SK is called a **superkey** of the relation schema *R*. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state *r* of *R* can have the same value for SK. Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key,* which has no redundancy. A **key** *k* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K′* that is not a superkey of *R* any more. Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This *uniqueness* property also applies to a superkey.

2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold. This *minimality* property is required for a key but is optional for a superkey.

Hence, a key is a superkey but not vice versa. A superkey may be a key (if it is minimal) or may not be a key (if it is not minimal). Consider the STUDENT relation of Figure 5.1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.[8] Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes together to have the uniqueness property.

The value of a key attribute can be used to identify uniquely each tuple in the relation. For example, the Ssn value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema. A key is determined from the meaning of the attributes, and the property is *time-invariant:* It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 5.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.[9]

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 5.4 has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 5.4. Notice that when a relation schema has several candidate keys,

---

[8]Note that Ssn is also a superkey.

[9]Names are sometimes used as keys, but then some artifact—such as appending an ordinal number—must be used to distinguish between persons with identical names.

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4
The CAR relation, with two candidate keys: License_number and Engine_serial_number.

the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys** and are not underlined.

Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

5.2.3 Relational Databases and Relational Database Schemas

The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section, we define a relational database and a relational database schema.

A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \ldots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state**[10] DB of S is a set of relation states DB = $\{r_1, r_2, \ldots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC. Figure 5.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. In each relation schema, the underlined attribute represents the primary key. Figure 5.6 shows a relational database state corresponding to the COMPANY schema. We will use this schema and database state in this chapter and in Chapters 4 through 6 for developing sample queries in different relational languages. (The data shown here is expanded and available for loading as a populated database from the Companion Website for the text, and can be used for the hands-on project exercises at the end of the chapters.)

When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is

---

[10]A relational database *state* is sometimes called a relational database *snapshot* or *instance*. However, as we mentioned earlier, we will not use the term *instance* since it also applies to single tuples.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure 5.5
Schema diagram for the
COMPANY relational
database schema.

called **not valid**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**

In Figure 5.5, the Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different real-world concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation of Figure 5.5: once in the role of the employee's SSN, and once in the role of the supervisor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. We present the SQL DDL in Sections 6.1 and 6.2.

Figure 5.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Integrity constraints are specified on a database schema and are expected to hold on *every valid database state* of that schema. In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 5.6, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define *referential integrity* more formally, first we define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.

2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ *or is NULL*. In the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2.

In this definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, first we must have a clear understanding of the meaning or role that each attribute or set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 5.6. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of

the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of Dno *can be NULL* if the employee does not belong to a department or will be assigned to a department later. For example, in Figure 5.6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department.

Notice that a foreign key can *refer to its own relation.* For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure 5.6 the tuple for employee 'John Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith'.

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 5.7 shows the schema in Figure 5.5 with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema (that is, specified as part of its definition) if we want the DBMS to enforce these constraints on

Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.
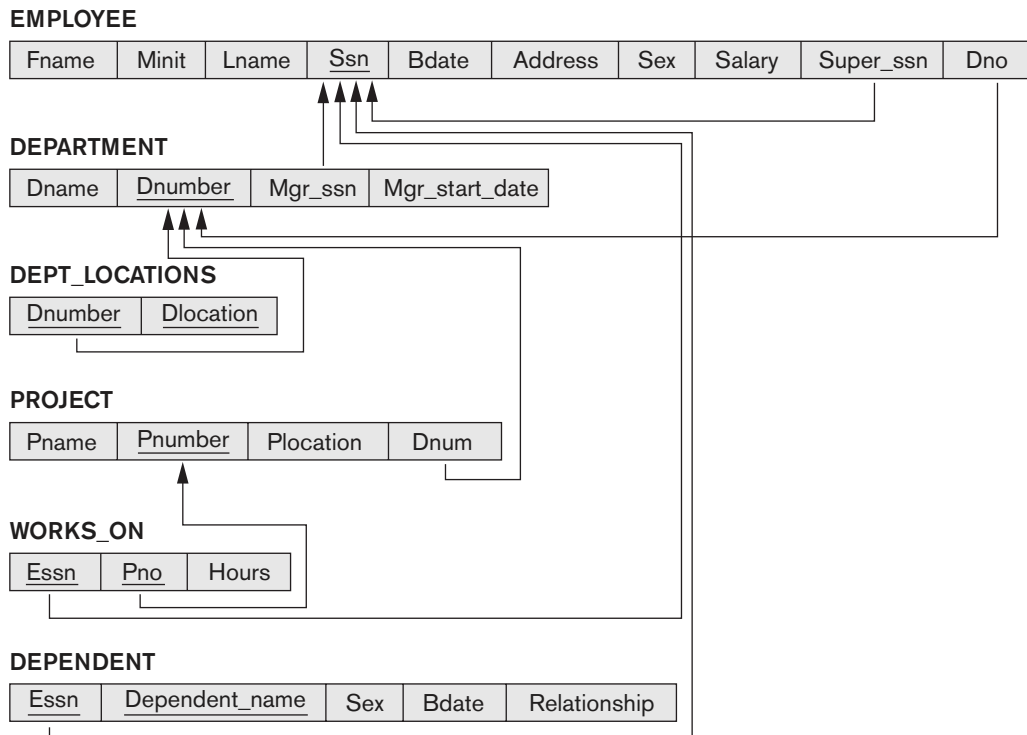
the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. In SQL, the CREATE TABLE statement of the SQL DDL allows the definition of primary key, unique key, NOT NULL, entity integrity, and referential integrity constraints, among other constraints (see Sections 6.1 and 6.2) .

5.2.5 Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. Another class of general constraints, sometimes called *semantic integrity constraints,* are not part of the DDL and have to be specified and enforced in a different way. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee's supervisor* and *the maximum number of hours an employee can work on all projects per week is 56.* Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**. Mechanisms called **triggers** and **assertions** can be used in SQL, through the CREATE ASSERTION and CREATE TRIGGER statements, to specify some of these constraints (see Chapter 7). It is more common to check for these types of constraints within the application programs than to use constraint specification languages because the latter are sometimes difficult and complex to use, as we discuss in Section 26.1.

The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy. Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.[11] An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically enforced by the application programs or specified using active rules and triggers, as we discuss in Section 26.1.

5.3 Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* The relational algebra operations, which can be used to specify **retrievals**, are discussed in detail in Chapter 8. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information. The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. The **result relation** becomes the answer to (or result of) the user's query. Chapter 8 also introduces the language

---

[11]State constraints are sometimes called *static constraints*, and transition constraints are sometimes called *dynamic constraints*.

called relational calculus, which is used to define a query declaratively without giving a specific order of operations.

In this section, we concentrate on the database **modification** or **update** operations. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records, respectively. **Insert** is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation. We use the database shown in Figure 5.6 for examples and discuss only domain constraints, key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 5.7. For each type of operation, we give some examples and discuss any constraints that each operation may violate.

5.3.1 The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R. Insert can violate any of the four types of constraints. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation $r(R)$. Entity integrity can be violated if any part of the primary key of the new tuple t is NULL. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

- *Operation*:
  Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
  Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

- *Operation*:
  Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.
  Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

- *Operation*:
  Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
  Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

■ *Operation*:
  Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
  Result: This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to *reject the insertion.* In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to *correct the reason for rejecting the insertion,* but this is *typically not used for violations caused by Insert*; rather, it is used more often in correcting violations for Delete and Update. In the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is provided. In operation 3, the DBMS could either ask the user to change the value of Dno to some valid value (or set it to NULL), or it could ask the user to insert a DEPARTMENT tuple with Dnumber = 7 and could accept the original insertion only after such an operation was accepted. Notice that in the latter case the insertion violation can **cascade** back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for Mgr_ssn that does not exist in the EMPLOYEE relation.

5.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

■ *Operation*:
  Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.
  Result: This deletion is acceptable and deletes exactly one tuple.

■ *Operation*:
  Delete the EMPLOYEE tuple with Ssn = '999887777'.
  Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

■ *Operation*:
  Delete the EMPLOYEE tuple with Ssn = '333445555'.
  Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation. The first option, called **restrict,** is to *reject the deletion.* The second option, called **cascade,** is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = '999887777'. A third option, called **set null** or **set default,** is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to

reference another default valid tuple. Notice that if a referencing attribute that causes a violation is *part of the primary key,* it *cannot* be set to NULL; otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = '333445555'. Tuples in EMPLOYEE with Super_ssn = '333445555' and the tuple in DEPARTMENT with Mgr_ssn = '333445555' can have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.

In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to *specify which of the options* applies in case of a violation of the constraint. We discuss how to specify these options in the SQL DDL in Chapter 6.

5.3.3 The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

- *Operation*:
  Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
  Result: Acceptable.
- *Operation*:
  Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
  Result: Acceptable.
- *Operation*:
  Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.
  Result: Unacceptable, because it violates referential integrity.
- *Operation*:
  Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.
  Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is *neither part of a primary key nor part of a foreign key* usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples. Hence, the issues discussed earlier in both Sections 5.3.1 (Insert) and 5.3.2 (Delete) come into play. If a foreign key attribute is modified, the

DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL). Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation. In fact, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to choose separate options to deal with a violation caused by Delete and a violation caused by Update (see Section 6.2).

5.3.4 The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*. A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations (to be discussed as part of relational algebra and calculus in Chapter 8, and as a part of the language SQL in Chapters 6 and 7) and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database. For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.

A large number of commercial applications running against relational databases in **online transaction processing (OLTP)** systems are executing transactions at rates that reach several hundred per second. Transaction processing concepts, concurrent execution of transactions, and recovery from failures will be discussed in Chapters 20 to 22.

5.4 Summary

In this chapter we presented the modeling concepts, data structures, and constraints provided by the relational model of data. We started by introducing the concepts of domains, attributes, and tuples. Then, we defined a relation schema as a list of attributes that describe the structure of a relation. A relation, or relation state, is a set of tuples that conforms to the schema.

Several characteristics differentiate relations from ordinary tables or files. The first is that a relation is not sensitive to the ordering of tuples. The second involves the ordering of attributes in a relation schema and the corresponding ordering of values within a tuple. We gave an alternative definition of relation that does not require ordering of attributes, but we continued to use the first definition, which requires attributes and tuple values to be ordered, for convenience. Then, we discussed values in tuples and introduced NULL values to represent missing or unknown information. We emphasized that NULL values should be avoided as much as possible.

We classified database constraints into inherent model-based constraints, explicit schema-based constraints, and semantic constraints or business rules. Then, we

discussed the schema constraints pertaining to the relational model, starting with domain constraints, then key constraints (including the concepts of superkey, key, and primary key), and the NOT NULL constraint on attributes. We defined relational databases and relational database schemas. Additional relational constraints include the entity integrity constraint, which prohibits primary key attributes from being NULL. We described the interrelation referential integrity constraint, which is used to maintain consistency of references among tuples from various relations.

The modification operations on the relational model are Insert, Delete, and Update. Each operation may violate certain types of constraints (refer to Section 5.3). Whenever an operation is applied, the resulting database state must be a valid state. Finally, we introduced the concept of a transaction, which is important in relational DBMSs because it allows the grouping of several database operations into a single atomic action on the database.

Review Questions

5.1. Define the following terms as they apply to the relational model of data: *domain*, *attribute*, *n-tuple*, *relation schema*, *relation state*, *degree of a relation*, *relational database schema*, and *relational database state*.

5.2. Why are tuples in a relation not ordered?

5.3. Why are duplicate tuples not allowed in a relation?

5.4. What is the difference between a key and a superkey?

5.5. Why do we designate one of the candidate keys of a relation to be the primary key?

5.6. Discuss the characteristics of relations that make them different from ordinary tables and files.

5.7. Discuss the various reasons that lead to the occurrence of NULL values in relations.

5.8. Discuss the entity integrity and referential integrity constraints. Why is each considered important?

5.9. Define *foreign key*. What is this concept used for?

5.10. What is a transaction? How does it differ from an Update operation?

Exercises

5.11. Suppose that each of the following Update operations is applied directly to the database state shown in Figure 5.6. Discuss *all* integrity constraints

violated by each operation, if any, and the different ways of enforcing these constraints.

a. Insert <'Robert', 'F', 'Scott', '943775543', '1972-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.

b. Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.

c. Insert <'Production', 4, '943775543', '2007-10-01'> into DEPARTMENT.

d. Insert <'677678989', NULL, '40.0'> into WORKS_ON.

e. Insert <'453453453', 'John', 'M', '1990-12-12', 'spouse'> into DEPENDENT.

f. Delete the WORKS_ON tuples with Essn = '333445555'.

g. Delete the EMPLOYEE tuple with Ssn = '987654321'.

h. Delete the PROJECT tuple with Pname = 'ProductX'.

i. Modify the Mgr_ssn and Mgr_start_date of the DEPARTMENT tuple with Dnumber = 5 to '123456789' and '2007-10-01', respectively.

j. Modify the Super_ssn attribute of the EMPLOYEE tuple with Ssn = '999887777' to '943775543'.

k. Modify the Hours attribute of the WORKS_ON tuple with Essn = '999887777' and Pno = 10 to '5.0'.

5.12. Consider the AIRLINE relational database schema shown in Figure 5.8, which describes a database for airline flight information. Each FLIGHT is identified by a Flight_number, and consists of one or more FLIGHT_LEGs with Leg_numbers 1, 2, 3, and so on. Each FLIGHT_LEG has scheduled arrival and departure times, airports, and one or more LEG_INSTANCEs— one for each Date on which the flight travels. FAREs are kept for each FLIGHT. For each FLIGHT_LEG instance, SEAT_RESERVATIONs are kept, as are the AIRPLANE used on the leg and the actual arrival and departure times and airports. An AIRPLANE is identified by an Airplane_id and is of a particular AIRPLANE_TYPE. CAN_LAND relates AIRPLANE_TYPEs to the AIRPORTs at which they can land. An AIRPORT is identified by an Airport_code. Consider an update for the AIRLINE database to enter a reservation on a particular flight or flight leg on a given date.

a. Give the operations for this update.

b. What types of constraints would you expect to check?

c. Which of these constraints are key, entity integrity, and referential integrity constraints, and which are not?

d. Specify all the referential integrity constraints that hold on the schema shown in Figure 5.8.

5.13. Consider the relation CLASS(Course#, Univ_Section#, Instructor_name, Semester, Building_code, Room#, Time_period, Weekdays, Credit_hours). This represents classes taught in a university, with unique Univ_section#s. Identify what you think should be various candidate keys, and write in your own words the conditions or assumptions under which each candidate key would be valid.

AIRPORT

Airport_code	Name	City	State

FLIGHT

Flight_number	Airline	Weekdays

FLIGHT_LEG

Flight_number	Leg_number	Departure_airport_code	Scheduled_departure_time
		Arrival_airport_code	Scheduled_arrival_time

LEG_INSTANCE

Flight_number	Leg_number	Date	Number_of_available_seats	Airplane_id
Departure_airport_code	Departure_time	Arrival_airport_code	Arrival_time	

FARE

Flight_number	Fare_code	Amount	Restrictions

AIRPLANE_TYPE

Airplane_type_name	Max_seats	Company

CAN_LAND

Airplane_type_name	Airport_code

AIRPLANE

Airplane_id	Total_number_of_seats	Airplane_type

SEAT_RESERVATION

Flight_number	Leg_number	Date	Seat_number	Customer_name	Customer_phone

Figure 5.8
The AIRLINE relational database schema.

5.14. Consider the following six relations for an order-processing database appli-
cation in a company:

CUSTOMER(Cust#, Cname, City)

ORDER(Order#, Odate, Cust#, Ord_amt)

ORDER_ITEM(Order#, Item#, Qty)

ITEM(<u>Item#</u>, Unit_price)

SHIPMENT(<u>Order#</u>, <u>Warehouse#</u>, Ship_date)

WAREHOUSE(<u>Warehouse#</u>, City)

Here, Ord_amt refers to total dollar amount of an order; Odate is the date the order was placed; and Ship_date is the date an order (or part of an order) is shipped from the warehouse. Assume that an order can be shipped from several warehouses. Specify the foreign keys for this schema, stating any assumptions you make. What other constraints can you think of for this database?

5.15. Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:

SALESPERSON(<u>Ssn</u>, Name, Start_year, Dept_no)

TRIP(Ssn, From_city, To_city, Departure_date, Return_date, <u>Trip_id</u>)

EXPENSE(<u>Trip_id</u>, <u>Account#</u>, Amount)

A trip can be charged to one or more accounts. Specify the foreign keys for this schema, stating any assumptions you make.

5.16. Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(<u>Ssn</u>, Name, Major, Bdate)

COURSE(<u>Course#</u>, Cname, Dept)

ENROLL(<u>Ssn</u>, <u>Course#</u>, <u>Quarter</u>, Grade)

BOOK_ADOPTION(<u>Course#</u>, <u>Quarter</u>, Book_isbn)

TEXT(<u>Book_isbn</u>, Book_title, Publisher, Author)

Specify the foreign keys for this schema, stating any assumptions you make.

5.17. Consider the following relations for a database that keeps track of automobile sales in a car dealership (OPTION refers to some optional equipment installed on an automobile):

CAR(<u>Serial_no</u>, Model, Manufacturer, Price)

OPTION(<u>Serial_no</u>, <u>Option_name</u>, Price)

SALE(<u>Salesperson_id</u>, <u>Serial_no</u>, Date, Sale_price)

SALESPERSON(<u>Salesperson_id</u>, Name, Phone)

First, specify the foreign keys for this schema, stating any assumptions you make. Next, populate the relations with a few sample tuples, and then give an example of an insertion in the SALE and SALESPERSON relations that *violates* the referential integrity constraints and of another insertion that does not.

5.18. Database design often involves decisions about the storage of attributes. For example, a Social Security number can be stored as one attribute or split into three attributes (one for each of the three hyphen-delineated groups of

numbers in a Social Security number—XXX-XX-XXXX). However, Social Security numbers are usually represented as just one attribute. The decision is based on how the database will be used. This exercise asks you to think about specific situations where dividing the SSN is useful.

5.19. Consider a STUDENT relation in a UNIVERSITY database with the following attributes (Name, Ssn, Local_phone, Address, Cell_phone, Age, Gpa). Note that the cell phone may be from a different city and state (or province) from the local phone. A possible tuple of the relation is shown below:

Name	Ssn	Local_phone	Address	Cell_phone	Age	Gpa
George Shaw William Edwards	123-45-6789	555-1234	123 Main St., Anytown, CA 94539	555-4321	19	3.75

a. Identify the critical missing information from the Local_phone and Cell_phone attributes. (*Hint*: How do you call someone who lives in a different state or province?)

b. Would you store this additional information in the Local_phone and Cell_phone attributes or add new attributes to the schema for STUDENT?

c. Consider the Name attribute. What are the advantages and disadvantages of splitting this field from one attribute into three attributes (first name, middle name, and last name)?

d. What general guideline would you recommend for deciding when to store information in a single attribute and when to split the information?

e. Suppose the student can have between 0 and 5 phones. Suggest two different designs that allow this type of information.

5.20. Recent changes in privacy laws have disallowed organizations from using Social Security numbers to identify individuals unless certain restrictions are satisfied. As a result, most U.S. universities cannot use SSNs as primary keys (except for financial data). In practice, Student_id, a unique identifier assigned to every student, is likely to be used as the primary key rather than SSN since Student_id can be used throughout the system.

a. Some database designers are reluctant to use generated keys (also known as *surrogate keys*) for primary keys (such as Student_id) because they are artificial. Can you propose any natural choices of keys that can be used to identify the student record in a UNIVERSITY database?

b. Suppose that you are able to guarantee uniqueness of a natural key that includes last name. Are you guaranteed that the last name will not change during the lifetime of the database? If last name can change, what solutions can you propose for creating a primary key that still includes last name but remains unique?

c. What are the advantages and disadvantages of using generated (surrogate) keys?

Selected Bibliography

The relational model was introduced by Codd (1970) in a classic paper. Codd also introduced relational algebra and laid the theoretical foundations for the relational model in a series of papers (Codd, 1971, 1972, 1972a, 1974); he was later given the Turing Award, the highest honor of the ACM (Association for Computing Machinery) for his work on the relational model. In a later paper, Codd (1979) discussed extending the relational model to incorporate more meta-data and semantics about the relations; he also proposed a three-valued logic to deal with uncertainty in relations and incorporating NULLs in the relational algebra. The resulting model is known as RM/T. Childs (1968) had earlier used set theory to model databases. Later, Codd (1990) published a book examining over 300 features of the relational data model and database systems. Date (2001) provides a retrospective review and analysis of the relational data model.

Since Codd's pioneering work, much research has been conducted on various aspects of the relational model. Todd (1976) describes an experimental DBMS called PRTV that directly implements the relational algebra operations. Schmidt and Swenson (1975) introduce additional semantics into the relational model by classifying different types of relations. Chen's (1976) entity–relationship model, which is discussed in Chapter 3, is a means to communicate the real-world semantics of a relational database at the conceptual level. Wiederhold and Elmasri (1979) introduce various types of connections between relations to enhance its constraints. Extensions of the relational model are discussed in Chapters 11 and 26. Additional bibliographic notes for other aspects of the relational model and its languages, systems, extensions, and theory are given in Chapters 6 to 9, 14, 15, 23, and 30. Maier (1983) and Atzeni and De Antonellis (1993) provide an extensive theoretical treatment of the relational data model.

This page intentionally left blank

Basic SQL

The SQL language may be considered one of the major reasons for the commercial success of relational databases. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, older network or hierarchical systems—to relational systems. This is because even if the users became dissatisfied with the particular relational DBMS product they were using, converting to another relational DBMS product was not expected to be too expensive and time-consuming because both systems followed the same language standards. In practice, of course, there are differences among various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if two relational DBMSs faithfully support the standard, then conversion between two systems should be simplified. Another advantage of having such a standard is that users may write statements in a database application program that can access data stored in two or more relational DBMSs without having to change the database sublanguage (SQL), as long as both/all of the relational DBMSs support standard SQL.

This chapter presents the *practical* relational model, which is based on the SQL standard for *commercial* relational DBMSs, whereas Chapter 5 presented the most important concepts underlying the *formal* relational data model. In Chapter 8 (Sections 8.1 through 8.5), we shall discuss the *relational algebra* operations, which are very important for understanding the types of requests that may be specified on a relational database. They are also important for query processing and optimization in a relational DBMS, as we shall see in Chapters 18 and 19. However, the relational algebra operations are too low-level for most commercial DBMS users because a query in relational algebra is written as a sequence of operations that, when executed, produces the required result. Hence, the user must specify how—that is, *in what order*—to execute the query operations. On the other hand, the SQL language

provides a higher-level *declarative* language interface, so the user only specifies *what* the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS. Although SQL includes some features from relational algebra, it is based to a greater extent on the *tuple relational calculus*, which we describe in Section 8.6. However, the SQL syntax is more user-friendly than either of the two formal languages.

The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. The standardization of SQL is a joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO), and the first SQL standard is called SQL-86 or SQL1. A revised and much expanded standard called SQL-92 (also referred to as SQL2) was subsequently developed. The next standard that is well-recognized is SQL:1999, which started out as SQL3. Additional updates to the standard are SQL:2003 and SQL:2006, which added XML features (see Chapter 13) among other updates to the language. Another update in 2008 incorporated more object database features into SQL (see Chapter 12), and a further update is SQL:2011. We will try to cover the latest version of SQL as much as possible, but some of the newer features are discussed in later chapters. It is also not possible to cover the language in its entirety in this text. It is important to note that when new features are added to SQL, it usually takes a few years for some of these features to make it into the commercial SQL DBMSs.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL *and* a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java or C/C++.[1]

The later SQL standards (starting with **SQL:1999**) are divided into a **core** specification plus specialized **extensions**. The core is supposed to be implemented by all RDBMS vendors that are SQL compliant. The extensions can be implemented as optional modules to be purchased independently for specific database applications such as data mining, spatial data, temporal data, data warehousing, online analytical processing (OLAP), multimedia data, and so on.

Because the subject of SQL is both important and extensive, we devote two chapters to its basic features. In this chapter, Section 6.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 6.2 presents how basic constraints such as key and referential integrity are specified. Section 6.3 describes the basic SQL constructs for

---

[1]Originally, SQL had statements for creating and dropping indexes on the files that represent relations, but these have been dropped from the SQL standard for some time.

specifying retrieval queries, and Section 6.4 describes the SQL commands for insertion, deletion, and update.

In Chapter 7, we will describe more complex SQL retrieval queries, as well as the ALTER commands for changing the schema. We will also describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database, and the concept of triggers, which is presented in more detail in Chapter 26. We discuss the SQL facility for defining views on the database in Chapter 7. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables.

Section 6.5 lists some SQL features that are presented in other chapters of the book; these include object-oriented features in Chapter 12, XML in Chapter 13, transaction control in Chapter 20, active databases (triggers) in Chapter 26, online analytical processing (OLAP) features in Chapter 29, and security/authorization in Chapter 30. Section 6.6 summarizes the chapter. Chapters 10 and 11 discuss the various database programming techniques for programming with SQL.

6.1 SQL Data Definition and Data Types

SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), types, and domains, as well as other constructs such as views, assertions, and triggers. Before we describe the relevant CREATE statements, we discuss schema and catalog concepts in Section 6.1.1 to place our discussion in perspective. Section 6.1.2 describes how tables are created, and Section 6.1.3 describes the most important data types available for attribute specification. Because the SQL specification is very large, we give a description of the most important features. Further details can be found in the various SQL standards documents (see end-of-chapter bibliographic notes).

6.1.1 Schema and Catalog Concepts in SQL

Early versions of SQL did not include the concept of a relational database schema; all tables (relations) were considered part of the same schema. The concept of an SQL schema was incorporated starting with SQL2 in order to group together tables and other constructs that belong to the same database application (in some systems, a *schema* is called a *database*). An **SQL schema** is identified by a **schema name** and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema. Schema **elements** include tables, types, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the

elements can be defined later. For example, the following statement creates a schema called COMPANY owned by the user with authorization identifier 'Jsmith'. Note that each statement in SQL ends with a semicolon.

CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmith';

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

In addition to the concept of a schema, SQL uses the concept of a **catalog**—a named collection of schemas.[2] Database installations typically have a default environment and schema, so when a user connects and logs in to that database installation, the user can refer directly to tables and other constructs within that schema without having to specify a particular schema name. A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the schemas in the catalog and all the element descriptors in these schemas. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as type and domain definitions.

6.1.2 The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and possibly attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command (see Chapter 7). Figure 6.1 shows sample data definition statements in SQL for the COMPANY relational database schema shown in Figure 3.7.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

CREATE TABLE COMPANY.EMPLOYEE

rather than

CREATE TABLE EMPLOYEE

as in Figure 6.1, we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.

The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the table and its rows are actually created

---

[2]SQL also includes the concept of a *cluster* of catalogs.

```
CREATE TABLE EMPLOYEE
       ( Fname                        VARCHAR(15)              NOT NULL,
         Minit                        CHAR,
         Lname                        VARCHAR(15)              NOT NULL,
         Ssn                          CHAR(9)                  NOT NULL,
         Bdate                        DATE,
         Address                      VARCHAR(30),
         Sex                          CHAR,
         Salary                       DECIMAL(10,2),
         Super_ssn                    CHAR(9),
         Dno                          INT                      NOT NULL,
       PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
       ( Dname                        VARCHAR(15)              NOT NULL,
         Dnumber                      INT                      NOT NULL,
         Mgr_ssn                      CHAR(9)                  NOT NULL,
         Mgr_start_date               DATE,
       PRIMARY KEY (Dnumber),
       UNIQUE (Dname),
       FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
       ( Dnumber                      INT                      NOT NULL,
         Dlocation                    VARCHAR(15)              NOT NULL,
       PRIMARY KEY (Dnumber, Dlocation),
       FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
       ( Pname                        VARCHAR(15)              NOT NULL,
         Pnumber                      INT                      NOT NULL,
         Plocation                    VARCHAR(15),
         Dnum                         INT                      NOT NULL,
       PRIMARY KEY (Pnumber),
       UNIQUE (Pname),
       FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
       ( Essn                         CHAR(9)                  NOT NULL,
         Pno                          INT                      NOT NULL,
         Hours                        DECIMAL(3,1)             NOT NULL,
       PRIMARY KEY (Essn, Pno),
       FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
       FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
       ( Essn                         CHAR(9)                  NOT NULL,
         Dependent_name               VARCHAR(15)              NOT NULL,
         Sex                          CHAR,
         Bdate                        DATE,
         Relationship                 VARCHAR(8),
       PRIMARY KEY (Essn, Dependent_name),
       FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Figure 6.1
SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 5.7.

and stored as a file by the DBMS. Base relations are distinguished from **virtual relations**, created through the CREATE VIEW statement (see Chapter 7), which may or may not correspond to an actual physical file. In SQL, the attributes in a base table are considered to be *ordered in the sequence in which they are specified* in the CREATE TABLE statement. However, rows (tuples) are not considered to be ordered within a table (relation).

It is important to note that in Figure 6.1, there are some *foreign keys that may cause errors* because they are specified either via circular references or because they refer to a table that has not yet been created. For example, the foreign key Super_ssn in the EMPLOYEE table is a circular reference because it refers to the EMPLOYEE table itself. The foreign key Dno in the EMPLOYEE table refers to the DEPARTMENT table, which has not been created yet. To deal with this type of problem, these constraints can be left out of the initial CREATE TABLE statement, and then added later using the ALTER TABLE statement (see Chapter 7). We displayed all the foreign keys in Figure 6.1 to show the complete COMPANY schema in one place.

6.1.3 Attribute Data Types and Domains in SQL

The basic **data types** available for attributes include numeric, character string, bit string, Boolean, date, and time.

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(i, j)—or DEC(i, j) or NUMERIC(i, j)—where i, the *precision*, is the total number of decimal digits and j, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

- **Character-string** data types are either fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters—or varying length—VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive* (a distinction is made between uppercase and lowercase).[3] For fixed-length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith' if needed. Padded blanks are generally ignored when strings are compared. For comparison purposes, strings are considered ordered in alphabetic (or lexicographic) order; if a string *str1* appears before another string *str2* in alphabetic order, then *str1* is considered to be less than *str2*.[4] There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings

---

[3]This is not the case with SQL keywords, such as CREATE or CHAR. With keywords, SQL is *case insensitive*, meaning that SQL treats uppercase and lowercase letters as equivalent in keywords.

[4]For nonalphabetic characters, there is a defined order.

in SQL. For example, 'abc' || 'XYZ' results in a single string 'abcXYZ'. Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

- **Bit-string** data types are either of fixed length n—BIT(n)—or varying length— BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B'10101'.[5] Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.

- A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN. We discuss the need for UNKNOWN and the three-valued logic in Chapter 7.

- The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by the SQL implementation. This implies that months should be between 1 and 12 and days must be between 01 and 31; furthermore, a day should be a valid day for the corresponding month. The < (less than) comparison can be used with dates or times—an *earlier* date is considered to be smaller than a later date, and similarly with time. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME; for example, DATE '2014-09-27' or TIME '09:12:47'. In addition, a data type TIME(i), where i is called *time fractional seconds precision*, specifies $i + 1$ additional positions for TIME—one position for an additional period (.) separator character, and i positions for specifying decimal fractions of a second. A TIME WITH TIME ZONE data type includes an additional six positions for specifying the *displacement* from the standard universal time zone, which is in the range +13:00 to –12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

Some additional data types are discussed below. The list of types discussed here is not exhaustive; different implementations have added more data types to SQL.

- A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single-quoted

---

[5]Bit strings whose length is a multiple of 4 can be specified in *hexadecimal* notation, where the literal string is preceded by X and each hexadecimal character represents 4 bits.

strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2014-09-27 09:12:47.648302'.

- ■ Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

The format of DATE, TIME, and TIMESTAMP can be considered as a special type of string. Hence, they can generally be used in string comparisons by being **cast** (or **coerced** or converted) into the equivalent strings.

It is possible to specify the data type of each attribute directly, as in Figure 6.1; alternatively, a domain can be declared, and the domain name can be used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE **AS** CHAR(9);

We can use SSN_TYPE in place of CHAR(9) in Figure 6.1 for the attributes Ssn and Super_ssn of EMPLOYEE, Mgr_ssn of DEPARTMENT, Essn of WORKS_ON, and Essn of DEPENDENT. A domain can also have an optional default specification via a DEFAULT clause, as we discuss later for attributes. Notice that domains may not be available in some implementations of SQL.

In SQL, there is also a **CREATE TYPE** command, which can be used to create user defined types or UDTs. These can then be used either as data types for attributes, or as the basis for creating tables. We shall discuss CREATE TYPE in detail in Chapter 12, because it is often used in conjunction with specifying object database features that have been incorporated into more recent versions of SQL.

6.2 Specifying Constraints in SQL

This section describes the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, restrictions on attribute domains and NULLs, and constraints on individual tuples within a relation using the CHECK clause. We discuss the specification of more general constraints, called assertions, in Chapter 7.

6.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the *primary key* of each relation, but it can be specified for any other attributes whose values are required not to be NULL, as shown in Figure 6.1.

It is also possible to define a *default value* for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any

```
CREATE TABLE EMPLOYEE
    ( … ,
      Dno            INT              NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                  ON DELETE SET NULL        ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE SET DEFAULT     ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
    ( … ,
      Mgr_ssn CHAR(9)         NOT NULL       DEFAULT '888665555',
      … ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                  ON DELETE SET DEFAULT     ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
    ( … ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE CASCADE         ON UPDATE CASCADE);
```

Figure 6.2
Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

new tuple if an explicit value is not provided for that attribute. Figure 6.2 illustrates an example of specifying a default manager for a new department and a default department for a new employee. If no default clause is specified, the default *default value* is NULL for attributes *that do not have* the NOT NULL constraint.

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition.[6] For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table (see Figure 6.1) to the following:

Dnumber INT **NOT NULL CHECK** (Dnumber $>$ 0 **AND** Dnumber $<$ 21);

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

CREATE DOMAIN D_NUM **AS** INTEGER
CHECK (D_NUM $>$ 0 **AND** D_NUM $<$ 21);

---

[6]The CHECK clause can also be used for other purposes, as we shall see.

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department numbers in Figure 6.1, such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

6.2.2 Specifying Key and Referential Integrity Constraints

Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them. Some examples to illustrate the specification of keys and referential integrity are shown in Figure 6.1.[7] The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a *single* attribute, the clause can follow the attribute directly. For example, the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure 6.1):

>   Dnumber INT **PRIMARY KEY**,

The **UNIQUE** clause specifies alternate (unique) keys, also known as candidate keys as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 6.1. The **UNIQUE** clause can also be specified directly for a unique key if it is a single attribute, as in the following example:

>   Dname VARCHAR(15) **UNIQUE**,

Referential integrity is specified via the **FOREIGN KEY** clause, as shown in Figure 6.1. As we discussed in Section 5.2.4, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is updated. The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken by attaching a **referential triggered action** clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. We illustrate this with the examples shown in Figure 6.2. Here, the database designer chooses ON DELETE SET NULL and ON UPDATE CASCADE for the foreign key Super_ssn of EMPLOYEE. This means that if the tuple for a *supervising employee* is *deleted*, the value of Super_ssn is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple. On the other hand, if the Ssn value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to Super_ssn for all employee tuples referencing the updated employee tuple.[8]

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the

---

[7]Key and referential integrity constraints were not included in early versions of SQL.

[8]Notice that the foreign key Super_ssn in the EMPLOYEE table is a circular reference and hence may have to be added later as a named constraint using the ALTER TABLE statement as we discussed at the end of Section 6.1.2.

referencing attribute for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples. It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema. As a general rule, the CASCADE option is suitable for "relationship" relations (see Section 9.1) , such as WORKS_ON; for relations that represent multivalued attributes, such as DEPT_LOCATIONS; and for relations that represent weak entity types, such as DEPENDENT.

6.2.3 Giving Names to Constraints

Figure 6.2 also illustrates how a constraint may be given a **constraint name**, following the keyword **CONSTRAINT**. The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint, as we discuss in Chapter 7. Giving names to constraints is optional. It is also possible to temporarily *defer* a constraint until the end of a transaction, as we shall discuss in Chapter 20 when we present transaction concepts.

6.2.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **row-based** constraints because they apply to each row *individually* and are checked whenever a row is inserted or modified. For example, suppose that the DEPARTMENT table in Figure 6.1 had an additional attribute Dept_create_date, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date.

> **CHECK** (Dept_create_date <= Mgr_start_date);

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL. We discuss this in Chapter 7 because it requires the full power of queries, which are discussed in Sections 6.3 and 7.1.

6.3 Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT** statement. The SELECT statement *is not the same as* the SELECT operation of relational algebra, which we shall discuss in Chapter 8. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually. We will use example queries specified on the schema of Figure 5.5 and will

refer to the sample database state shown in Figure 5.6 to show the results of some of these queries. In this section, we present the features of SQL for *simple retrieval queries*. Features of SQL for specifying more complex retrieval queries are presented in Section 7.1.

Before proceeding, we must point out an *important distinction* between the practical SQL model and the formal relational model discussed in Chapter 5: SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an **SQL** table is not a *set of tuples*, because a set does not allow two identical members; rather, it is a **multiset** (sometimes called a *bag*) of tuples. Some SQL relations are *constrained to be sets* because a key constraint has been declared or because the DISTINCT option has been used with the SELECT statement (described later in this section). We should be aware of this distinction as we discuss the examples.

6.3.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

Queries in SQL can be very complex. We will start with simple queries, and then progress to more complex ones in a step-by-step manner. The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block**, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:[9]

SELECT	\<attribute list\>
FROM	\<table list\>
WHERE	\<condition\>;

where

- \<attribute list\> is a list of attribute names whose values are to be retrieved by the query.
- \<table list\> is a list of the relation names required to process the query.
- \<condition\> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively, and to the C/C++ programming language operators =, <, <=, >, >=, and !=. The main syntactic difference is the *not equal* operator. SQL has additional comparison operators that we will present gradually.

We illustrate the basic SELECT statement in SQL with some sample queries. The queries are labeled here with the same query numbers used in Chapter 8 for easy cross-reference.

---

[9]The SELECT and FROM clauses are required in all SQL queries. The WHERE is optional (see Section 6.3.3).

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

Q0:   **SELECT**   Bdate, Address
        FROM   EMPLOYEE
        WHERE   Fname = 'John' **AND** Minit = 'B' **AND** Lname = 'Smith';

This query involves only the EMPLOYEE relation listed in the FROM clause. The query *selects* the individual EMPLOYEE tuples that satisfy the condition of the WHERE clause, then *projects* the result on the Bdate and Address attributes listed in the SELECT clause.

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the **projection attributes** in relational algebra (see Chapter 8) and the WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition** in relational algebra. Figure 6.3(a) shows the result of query Q0 on the database of Figure 5.6.

We can think of an implicit **tuple variable** or *iterator* in the SQL query ranging or *looping* over each individual tuple in the EMPLOYEE table and evaluating the condition in the WHERE clause. Only those tuples that satisfy the condition—that is, those tuples for which the condition evaluates to TRUE after substituting their corresponding attribute values—are selected.

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

Q1:   **SELECT**   Fname, Lname, Address
        FROM   EMPLOYEE, DEPARTMENT
        WHERE   Dname = 'Research' **AND** Dnumber = Dno;

In the WHERE clause of Q1, the condition Dname = 'Research' is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE. The result of query Q1 is shown in Figure 6.3(b). In general, any number of selection and join conditions may be specified in a single SQL query.

A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join** query. The next example is a select-project-join query with *two* join conditions.

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

Q2:   **SELECT**   Pnumber, Dnum, Lname, Address, Bdate
        FROM   PROJECT, DEPARTMENT, EMPLOYEE
        WHERE   Dnum = Dnumber **AND** Mgr_ssn = Ssn **AND**
                      Plocation = 'Stafford'

Figure 6.3
Results of SQL queries when applied to the COMPANY database state shown
in Figure 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C.

(a)

Bdate	Address
1965-01-09	731 Fondren, Houston, TX

(b)

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

(c)

Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

(f)

Ssn	Dname
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

(d)

E.Fname	E.Lname	S.Fname	S.Lname
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

(e)

E.Fname
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(g)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn = Ssn relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a *combination* of one project, one department (that controls the project), and one employee (that manages the department). The projection attributes are used to choose the attributes to be displayed from each combined tuple. The result of query Q2 is shown in Figure 6.3(c).

6.3.2 Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in *different tables.* If this is the case, and a multitable query refers to two or more attributes with the same name, we *must* **qualify** the attribute name with the relation name to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figures 5.5 and 5.6 the Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes Name and Dnumber in Q1A to specify which ones we are referring to, because the same attribute names are used in both relations:

Q1A:	**SELECT**	Fname, EMPLOYEE.Name, Address
	FROM	EMPLOYEE, DEPARTMENT
	WHERE	DEPARTMENT.Name = 'Research' **AND**
		DEPARTMENT.Dnumber = EMPLOYEE.Dnumber;

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. Q1 can be rewritten as Q1′ below with fully qualified attribute names. We can also rename the table names to shorter names by creating an *alias* for each table name to avoid repeated typing of long table names (see Q8 below).

Q1′:	**SELECT**	EMPLOYEE.Fname, EMPLOYEE.LName,
		EMPLOYEE.Address
	FROM	EMPLOYEE, DEPARTMENT
	WHERE	DEPARTMENT.DName = 'Research' **AND**
		DEPARTMENT.Dnumber = EMPLOYEE.Dno;

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice, as in the following example.

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

Q8:	**SELECT**	E.Fname, E.Lname, S.Fname, S.Lname
	FROM	EMPLOYEE **AS** E, EMPLOYEE **AS** S
	WHERE	E.Super_ssn = S.Ssn;

In this case, we are required to declare alternative relation names E and S, called **aliases** or **tuple variables**, for the EMPLOYEE relation. An alias can follow the keyword **AS**, as shown in Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q8. It is also possible to **rename** the relation attributes within the query in SQL by giving them aliases. For example, if we write

> EMPLOYEE **AS** E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)

in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on.

In Q8, we can think of E and S as two *different copies* of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees or subordinates; the second, S, represents employees in the role of supervisors. We can now join the two copies. Of course, in reality there is *only one* EMPLOYEE relation, and the join condition is meant to join the relation with itself by matching the tuples that satisfy the join condition E.Super_ssn = S.Ssn. Notice that this is an example of a one-level recursive query, as we will discuss in Section 8.4.2. In earlier versions of SQL, it was not possible to specify a general recursive query, with an unknown number of levels, in a single SQL statement. A construct for specifying recursive queries has been incorporated into SQL:1999 (see Chapter 7).

The result of query Q8 is shown in Figure 6.3(d). Whenever one or more aliases are given to a relation, we can use these names to represent different references to that same relation. This permits multiple references to the same relation within a query.

We can use this alias-naming or **renaming** mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once. In fact, this practice is recommended since it results in queries that are easier to comprehend. For example, we could specify query Q1 as in Q1B:

Q1B:	**SELECT**	E.Fname, E.LName, E.Address
	FROM	EMPLOYEE **AS** E, DEPARTMENT **AS** D
	WHERE	D.DName = 'Research' **AND** D.Dnumber = E.Dno;

6.3.3 Unspecified WHERE Clause and Use of the Asterisk

We discuss two more features of SQL here. A *missing* WHERE clause indicates no condition on tuple selection; hence, *all tuples* of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected. For example, Query 9 selects all EMPLOYEE Ssns (Figure 6.3(e)), and Query 10 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not (Figure 6.3(f)).

Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

| Q9: | **SELECT** | Ssn |
| | **FROM** | EMPLOYEE; |

| Q10: | **SELECT** | Ssn, Dname |
| | **FROM** | EMPLOYEE, DEPARTMENT; |

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result. Notice that Q10 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra (see Chapter 8). If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the actual CROSS PRODUCT (except for duplicate elimination, if any).

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes.* The * can also be prefixed by the relation name or alias; for example, EMPLOYEE.* refers to all attributes of the EMPLOYEE table.

Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 6.3(g)), query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department, and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

Q1C:	**SELECT**	*
	FROM	EMPLOYEE
	WHERE	Dno = 5;

Q1D:	**SELECT**	*
	FROM	EMPLOYEE, DEPARTMENT
	WHERE	Dname = 'Research' **AND** Dno = Dnumber;

| Q10A: | **SELECT** | * |
| | **FROM** | EMPLOYEE, DEPARTMENT; |

6.3.4 Tables as Sets in SQL

As we mentioned earlier, SQL usually treats a table not as a set but rather as a **multiset**; *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function (see Section 7.1.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

Figure 6.4
Results of additional
SQL queries when
applied to the
COMPANY database
state shown in
Figure 5.6. (a) Q11.
(b) Q11A. (c) Q16.
(d) Q18.

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

(c)

Fname	Lname

(d)

Fname	Lname
James	Borg

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.[10] If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither ALL nor DISTINCT—as in our previous examples—is equivalent to SELECT ALL. For example, Q11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in Figure 6.4(a). If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword **DISTINCT** as in Q11A, we accomplish this, as shown in Figure 6.4(b).

> **Query 11.** Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).
>
> Q11: **SELECT** **ALL** Salary
>   **FROM** EMPLOYEE;
>
> Q11A: **SELECT** **DISTINCT** Salary
>   **FROM** EMPLOYEE;

SQL has directly incorporated some of the set operations from mathematical *set theory*, which are also part of relational algebra (see Chapter 8). There are set union (**UNION**), set difference (**EXCEPT**),[11] and set intersection (**INTERSECT**) operations. The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result.* These set operations apply only to *type-compatible relations*, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

---

[10]In general, an SQL table is not required to have a key, although in most cases there will be one.

[11]In some systems, the keyword MINUS is used for the set difference operation instead of EXCEPT.

(a)

R
A
a1
a2
a2
a3

S
A
a1
a2
a4
a5

(b)

T
A
a1
a1
a2
a2
a2
a3
a4
a5

(c)

T
A
a2
a3

(d)

T
A
a1
a2

Figure 6.5
The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A)UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

Q4A:	(**SELECT**	**DISTINCT** Pnumber
	FROM	PROJECT, DEPARTMENT, EMPLOYEE
	WHERE	Dnum = Dnumber **AND** Mgr_ssn = Ssn
		AND Lname = 'Smith')
	UNION	
	(**SELECT**	**DISTINCT** Pnumber
	FROM	PROJECT, WORKS_ON, EMPLOYEE
	WHERE	Pnumber = Pno **AND** Essn = Ssn
		AND Lname = 'Smith');

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project. Notice that if several employees have the last name 'Smith', the project names involving any of them will be retrieved. Applying the UNION operation to the two SELECT queries gives the desired result.

SQL also has corresponding multiset operations, which are followed by the keyword **ALL** (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated). The behavior of these operations is illustrated by the examples in Figure 6.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

6.3.5 Substring Pattern Matching and Arithmetic Operators

In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the **LIKE** comparison operator. This can be used for string **pattern matching**. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query.

Query 12. Retrieve all employees whose address is in Houston, Texas.

Q12:   **SELECT**   Fname, Lname
        FROM   EMPLOYEE
        WHERE   Address **LIKE** '%Houston,TX%';

To retrieve all employees who were born during the 1970s, we can use Query Q12A. Here, '7' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a place-holder for an arbitrary character.

Query 12A. Find all employees who were born during the 1950s.

Q12:   **SELECT**   Fname, Lname
        FROM   EMPLOYEE
        WHERE   Bdate **LIKE** '_ _ 7 _ _ _ _ _ _';

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character*, which is specified after the string using the keyword ESCAPE. For example, 'AB_CD\%EF' **ESCAPE** '\' represents the lit-eral string 'AB_CD%EF' because \ is specified as the escape character. Any charac-ter not used in the string can be chosen as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (' ') if they are to be included in a string because they are used to begin and end strings. If an apostrophe (') is needed, it is represented as two consecutive apostrophes ('') so that it will not be interpreted as ending the string. Notice that substring comparison implies that attribute values are not atomic (indivisible) values, as we had assumed in the formal relational model (see Section 5.1) .

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (−), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10% raise; we can issue Query 13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

Query 13. Show the resulting salaries if every employee working on the 'ProductX' project is given a 10% raise.

Q13:   **SELECT**   E.Fname, E.Lname, 1.1 * E.Salary **AS** Increased_sal
        FROM   EMPLOYEE **AS** E, WORKS_ON **AS** W, PROJECT **AS** P
        WHERE   E.Ssn = W.Essn **AND** W.Pno = P.Pnumber **AND**
                      P.Pname = 'ProductX';

For string data types, the concatenate operator || can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (−) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator, which can be used for convenience, is **BETWEEN**, which is illustrated in Query 14.

Query 14. Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

Q14:   **SELECT**   *
          FROM   EMPLOYEE
          WHERE   (Salary **BETWEEN** 30000 **AND** 40000) **AND** Dno = 5;

The condition (Salary **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the condition ((Salary >= 30000) **AND** (Salary <= 40000)).

6.3.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause. This is illustrated by Query 15.

Query 15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

Q15:   **SELECT**   D.Dname, E.Lname, E.Fname, P.Pname
          FROM   DEPARTMENT **AS** D, EMPLOYEE **AS** E, WORKS_ON **AS** W,
                        PROJECT **AS** P
          WHERE   D.Dnumber = E.Dno **AND** E.Ssn = W.Essn **AND** W.Pno =
                        P.Pnumber
          ORDER BY   D.Dname, E.Lname, E.Fname;

The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values. The keyword **ASC** can be used to specify ascending order explicitly. For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause of Q15 can be written as

   ORDER BY D.Dname **DESC**, E.Lname **ASC**, E.Fname **ASC**

6.3.7 Discussion and Summary of Basic SQL Retrieval Queries

A *simple* retrieval query in SQL can consist of up to four clauses, but only the first two—SELECT and FROM—are mandatory. The clauses are specified in the following order, with the clauses between square brackets […] being optional:

   SELECT   <attribute list>
   FROM   <table list>
   [**WHERE**   <condition>]
   [**ORDER BY**   <attribute list>];

The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including

join conditions if needed. ORDER BY specifies an order for displaying the results of a query. Two additional clauses GROUP BY and HAVING will be described in Section 7.1.8.

In Chapter 7, we will present more complex features of SQL retrieval queries. These include the following: nested queries that allow one query to be included as part of another query; aggregate functions that are used to provide summaries of the information in the tables; two additional clauses (GROUP BY and HAVING) that can be used to provide additional power to aggregate functions; and various types of joins that can combine records from various tables in different ways.

6.4 INSERT, DELETE, and UPDATE Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

6.4.1 The INSERT Command

In its simplest form, INSERT is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation shown in Figure 5.5 and specified in the CREATE TABLE EMPLOYEE … command in Figure 6.1, we can use U1:

U1:	INSERT INTO	EMPLOYEE
	VALUES	('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification *and* no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be *left out*. For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes, we can use U1A:

U1A:	INSERT INTO	EMPLOYEE (Fname, Lname, Dno, Ssn)
	VALUES	('Richard', 'Marini', 4, '653298653');

Attributes not specified in U1A are set to their DEFAULT or to NULL, and the values are listed in the same order as the *attributes are listed in the INSERT* command itself. It is also possible to insert into a relation *multiple tuples* separated by commas in a single INSERT command. The attribute values forming *each tuple* are enclosed in parentheses.

A DBMS that fully implements SQL should support and enforce all the integrity constraints that can be specified in the DDL. For example, if we issue the command in U2 on the database shown in Figure 5.6, the DBMS should *reject* the operation because no DEPARTMENT tuple exists in the database with Dnumber = 2. Similarly, U2A would be *rejected* because no Ssn value is provided and it is the primary key, which cannot be NULL.

U2:	**INSERT INTO**	EMPLOYEE (Fname, Lname, Ssn, Dno)
	VALUES	('Robert', 'Hatcher', '980760540', 2);
	(U2 is rejected if referential integrity checking is provided by DBMS.)	

U2A:	**INSERT INTO**	EMPLOYEE (Fname, Lname, Dno)
	VALUES	('Robert', 'Hatcher', 5);
	(U2A is rejected if NOT NULL checking is provided by DBMS.)	

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query.* For example, to create a temporary table that has the employee last name, project name, and hours per week for each employee working on a project, we can write the statements in U3A and U3B:

U3A:	**CREATE TABLE**	WORKS_ON_INFO
	(Emp_name	VARCHAR(15),
	Proj_name	VARCHAR(15),
	Hours_per_week	DECIMAL(3,1));

U3B:	**INSERT INTO**	WORKS_ON_INFO (Emp_name, Proj_name, Hours_per_week)
	SELECT	E.Lname, P.Pname, W.Hours
	FROM	PROJECT P, WORKS_ON W, EMPLOYEE E
	WHERE	P.Pnumber = W.Pno **AND** W.Essn = E.Ssn;

A table WORKS_ON_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B. We can now query WORKS_ON_INFO as we would any other relation; when we do not need it anymore, we can remove it by using the DROP TABLE command (see Chapter 7). Notice that the WORKS_ON_INFO table may not be up to date; that is, if we update any of the PROJECT, WORKS_ON, or EMPLOYEE relations after issuing U3B, the information in WORKS_ON_INFO *may become outdated.* We have to create a view (see Chapter 7) to keep such a table up to date.

Most DBMSs have *bulk loading* tools that allow a user to load formatted data from a file into a table without having to write a large number of INSERT commands. The user can also write a program to read each record in the file, format it as a row in the table, and insert it using the looping constructs of a programming language (see Chapters 10 and 11, where we discuss database programming techniques).

Another variation for loading data is to create a new table TNEW that has the same attributes as an existing table T, and load some of the data currently in T into TNEW. The syntax for doing this uses the LIKE clause. For example, if we

want to create a table D5EMPS with a similar structure to the EMPLOYEE table and load it with the rows of employees who work in department 5, we can write the following SQL:

CREATE TABLE	D5EMPS **LIKE** EMPLOYEE
(**SELECT**	E.*
FROM	EMPLOYEE **AS** E
WHERE	E.Dno = 5) **WITH DATA**;

The clause WITH DATA specifies that the table will be created and loaded with the data specified in the query, although in some implementations it may be left out.

6.4.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL (see Section 6.2.2).[12] Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition (see Chapter 7). The DELETE commands in U4A to U4D, if applied independently to the database state shown in Figure 5.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

U4A:	**DELETE FROM**	EMPLOYEE
	WHERE	Lname = 'Brown';
U4B:	**DELETE FROM**	EMPLOYEE
	WHERE	Ssn = '123456789';
U4C:	**DELETE FROM**	EMPLOYEE
	WHERE	Dno = 5;
U4D:	**DELETE FROM**	EMPLOYEE;

6.4.3 The UPDATE Command

The **UPDATE** command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity

---

[12]Other actions can be automatically applied through triggers (see Section 26.1) and other mechanisms.

constraints of the DDL (see Section 6.2.2). An additional **SET** clause in the UPDATE command specifies the attributes to be modified and their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

U5:	**UPDATE**	PROJECT
	SET	Plocation = 'Bellaire', Dnum = 5
	WHERE	Pnumber = 10;

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10% raise in salary, as shown in U6. In this request, the modified Salary value depends on the original Salary value in each tuple, so two references to the Salary attribute are needed. In the SET clause, the reference to the Salary attribute on the right refers to the old Salary value *before modification*, and the one on the left refers to the new Salary value *after modification*:

U6:	**UPDATE**	EMPLOYEE
	SET	Salary = Salary * 1.1
	WHERE	Dno = 5;

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

6.5 Additional Features of SQL

SQL has a number of additional features that we have not described in this chapter but that we discuss elsewhere in the book. These are as follows:

- In Chapter 7, which is a continuation of this chapter, we will present the following SQL features: various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, case statements, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.

- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules). We discuss these techniques in Chapter 10. We also describe how to access SQL databases through the Java programming language using JDBC and SQLJ.

- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We called these commands a *storage definition language* (*SDL*) in Chapter 2. Earlier versions of SQL had commands for **creating indexes**, but these were removed from the

language because they were not at the conceptual schema level. Many systems still have the CREATE INDEX commands; but they require a special privilege. We describe this in Chapter 17.

■ SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes. We discuss these commands in Chapter 20 after we discuss the concept of transactions in more detail.

■ SQL has language constructs for specifying the *granting and revoking of privileges* to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or UPDATE—to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called **GRANT** and **REVOKE**—are discussed in Chapter 20, where we discuss database security and authorization.

■ SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates. We discuss these features in Section 26.1, where we discuss active database concepts.

■ SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational**. Capabilities such as creating complex-structured attributes, specifying abstract data types (called **UDT**s or user-defined types) for attributes and tables, creating **object identifiers** for referencing tuples, and specifying **operations** on types are discussed in Chapter 12.

■ SQL and relational databases can interact with new technologies such as XML (see Chapter 13) and OLAP/data warehouses (Chapter 29).

6.6 Summary

In this chapter, we introduced the SQL database language. This language and its variations have been implemented as interfaces to many commercial relational DBMSs, including Oracle's Oracle; ibm's DB2; Microsoft's SQL Server; and many other systems including Sybase and INGRES. Some open source systems also provide SQL, such as MySQL and PostgreSQL. The original version of SQL was implemented in the experimental DBMS called SYSTEM R, which was developed at IBM Research. SQL is designed to be a comprehensive language that includes statements for data definition, queries, updates, constraint specification, and view definition. We discussed the following features of SQL in this chapter: the data definition commands for creating tables, SQL basic data types, commands for constraint specification, simple retrieval queries, and database update commands. In the next chapter, we will present the following features of SQL: complex retrieval queries; views; triggers and assertions; and schema modification commands.

Review Questions

6.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 3? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

6.2. List the data types that are allowed for SQL attributes.

6.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 3? What about referential triggered actions?

6.4. Describe the four clauses in the syntax of a simple SQL retrieval query. Show what type of constructs can be specified in each of the clauses. Which are required and which are optional?

Exercises

6.5. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. What are the referential integrity constraints that should hold on the schema? Write appropriate SQL DDL statements to define the database.

6.6. Repeat Exercise 6.5, but use the AIRLINE database schema of Figure 5.8.

6.7. Consider the LIBRARY relational database schema shown in Figure 6.6. Choose the appropriate action (reject, cascade, set to NULL, set to default) for each referential integrity constraint, both for the *deletion* of a referenced tuple and for the *update* of a primary key attribute value in a referenced tuple. Justify your choices.

6.8. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 6.6. Specify the keys and referential triggered actions.

6.9. How can the key and foreign key constraints be enforced by the DBMS? Is the enforcement technique you suggest difficult to implement? Can the constraint checks be executed efficiently when updates are applied to the database?

6.10. Specify the following queries in SQL on the COMPANY relational database schema shown in Figure 5.5. Show the result of each query if it is applied to the COMPANY database in Figure 5.6.

a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.

b. List the names of all employees who have a dependent with the same first name as themselves.

c. Find the names of all employees who are directly supervised by 'Franklin Wong'.

BOOK

Book_id	Title	Publisher_name

BOOK_AUTHORS

Book_id	Author_name

PUBLISHER

Name	Address	Phone

BOOK_COPIES

Book_id	Branch_id	No_of_copies

BOOK_LOANS

Book_id	Branch_id	Card_no	Date_out	Due_date

LIBRARY_BRANCH

Branch_id	Branch_name	Address

BORROWER

Card_no	Name	Address	Phone

Figure 6.6
A relational database schema for a LIBRARY database.

6.11. Specify the updates of Exercise 3.11 using the SQL update commands.

6.12. Specify the following queries in SQL on the database schema of Figure 1.2.

a. Retrieve the names of all senior students majoring in 'cs' (computer science).

b. Retrieve the names of all courses taught by Professor King in 2007 and 2008.

c. For each section taught by Professor King, retrieve the course number, semester, year, and number of students who took the section.

d. Retrieve the name and transcript of each senior student (Class = 4) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.

6.13. Write SQL update statements to do the following on the database schema shown in Figure 1.2.

    a. Insert a new student, <'Johnson', 25, 1, 'Math'>, in the database.

    b. Change the class of student 'Smith' to 2.

    c. Insert a new course, <'Knowledge Engineering', 'cs4390', 3, 'cs'>.

    d. Delete the record for the student whose name is 'Smith' and whose student number is 17.

6.14. Design a relational database schema for a database application of your choice.

    a. Declare your relations using the SQL DDL.

    b. Specify a number of queries in SQL that are needed by your database application.

    c. Based on your expected use of the database, choose some attributes that should have indexes specified on them.

    d. Implement your database, if you have a DBMS that supports SQL.

6.15. Consider that the EMPLOYEE table's constraint EMPSUPERFK as specified in Figure 6.2 is changed to read as follows:

      CONSTRAINT EMPSUPERFK
        FOREIGN KEY (Super_ssn) **REFERENCES** EMPLOYEE(Ssn)
                    ON DELETE CASCADE **ON UPDATE** CASCADE,

Answer the following questions:

    a. What happens when the following command is run on the database state shown in Figure 5.6?

      DELETE EMPLOYEE **WHERE** Lname = 'Borg'

    b. Is it better to CASCADE or SET NULL in case of EMPSUPERFK constraint ON DELETE?

6.16. Write SQL statements to create a table EMPLOYEE_BACKUP to back up the EMPLOYEE table shown in Figure 5.6.

Selected Bibliography

The SQL language, originally named SEQUEL, was based on the language SQUARE (Specifying Queries as Relational Expressions) described by Boyce et al. (1975). The syntax of SQUARE was modified into SEQUEL (Chamberlin & Boyce, 1974) and then into SEQUEL 2 (Chamberlin et al., 1976), on which SQL is based. The original implementation of SEQUEL was done at IBM Research, San Jose, California. We will give additional references to various aspects of SQL at the end of Chapter 7.

This page intentionally left blank

More SQL: Complex Queries, Triggers, Views, and Schema Modification

This chapter describes more advanced features of the SQL language for relational databases. We start in Section 7.1 by presenting more complex features of SQL retrieval queries, such as nested queries, joined tables, outer joins, aggregate functions, and grouping, and case statements. In Section 7.2, we describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers and the CREATE TRIGGER statement, which will be presented in more detail in Section 26.1 when we present the principles of active databases. Then, in Section 7.3, we describe the SQL facility for defining views on the database. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables. Section 7.4 introduces the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints. Section 7.5 is the chapter summary.

This chapter is a continuation of Chapter 6. The instructor may skip parts of this chapter if a less detailed introduction to SQL is intended.

7.1 More Complex SQL Retrieval Queries

In Section 6.3, we described some basic types of retrieval queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex retrievals from the database. We discuss several of these features in this section.

7.1.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 5.1.2 that NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown* (value exists but is not known, or it is not known whether or not the value exists), value *not available* (value exists but is purposely withheld), or value *not applicable* (the attribute does not apply to this tuple or is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database. An example of the other case of unknown would be NULL for a person's home phone because it is not known whether or not the person has a home phone.

2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.

3. **Not applicable attribute.** An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish among the different meanings of NULL.

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a record with NULL in one of its attributes is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 7.1 shows the resulting values.

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	**AND**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	**OR**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	**NOT**			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

In Tables 7.1(a) and 7.1(b), the rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE, or UNKNOWN. The result of combining the two values using the AND logical connective is shown by the entries in Table 7.1(a). Table 7.1(b) shows the result of using the OR logical connective. For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN. Table 7.1(c) shows the result of the NOT logical operation. Notice that in standard Boolean logic, only TRUE or FALSE values are permitted; there is no UNKNOWN value.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. However, there are exceptions to that rule for certain operations, such as outer joins, as we shall see in Section 7.1.6.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or IS **NOT**. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN; see Section 7.1.6). Query 18 illustrates NULL comparison by retrieving any employees who do not have a supervisor.

> **Query 18.** Retrieve the names of all employees who do not have supervisors.

Q18:   **SELECT**   Fname, Lname
            FROM   EMPLOYEE
            WHERE   Super_ssn **IS** NULL;

7.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within another SQL query. That other query is called the **outer query**. These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed. Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the comparison operator **IN**, which compares a value v with a set (or multiset) of values V and evaluates to **TRUE** if v is one of the elements in V.

In Q4A, the first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, whereas the second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker. In the outer query, we use the **OR** logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

```
Q4A:    SELECT      DISTINCT Pnumber
        FROM        PROJECT
        WHERE       Pnumber IN
                    ( SELECT      Pnumber
                      FROM        PROJECT, DEPARTMENT, EMPLOYEE
                      WHERE       Dnum = Dnumber AND
                                  Mgr_ssn = Ssn AND Lname = 'Smith' )
                    OR
                    Pnumber IN
                    ( SELECT      Pno
                      FROM        WORKS_ON, EMPLOYEE
                      WHERE       Essn = Ssn AND Lname = 'Smith' );
```

If a nested query returns a single attribute *and* a single tuple, the query result will be a single (**scalar**) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a **table** (relation), which is a set or multiset of tuples.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       (Pno, Hours) IN    ( SELECT      Pno, Hours
                                  FROM        WORKS_ON
                                  WHERE       Essn = '123456789' );
```

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset v (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to *some value* in the set V and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition ($v >$ ALL V) returns TRUE if the value v is greater than *all* the values in the set (or multiset) V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT      Lname, Fname
FROM        EMPLOYEE
WHERE       Salary > ALL    ( SELECT      Salary
                              FROM        EMPLOYEE
                              WHERE       Dno = 5 );
```

Notice that this query can also be specified using the MAX aggregate function (see Section 7.1.7).

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query,* and another in a relation in the FROM clause of the *nested query.* The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query**. For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we specify and refer to an *alias* (tuple variable) for that relation. These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16.

> **Query 16.** Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

> Q16:   **SELECT**   E.Fname, E.Lname
>   **FROM**   EMPLOYEE **AS** E
>   **WHERE**   E.Ssn **IN**   (**SELECT**   D.Essn
>   **FROM**   DEPENDENT **AS** D
>   **WHERE**   E.Fname = D.Dependent_name
>   **AND** E.Sex = D.Sex);

In the nested query of Q16, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not *have to* qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fname and Ssn, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities, as illustrated in Q16.

7.1.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**. We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query.* For example, we can think of Q16 as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the Essn values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the Ssn value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q16 may be written as in Q16A:

Q16A:	**SELECT**	E.Fname, E.Lname
	FROM	EMPLOYEE **AS** E, DEPENDENT **AS** D
	WHERE	E.Ssn = D.Essn **AND** E.Sex = D.Sex
		AND E.Fname = D.Dependent_name;

7.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE; hence, they can be used in a WHERE clause condition. The EXISTS function in SQL is used to check whether the result of a nested query is *empty* (contains no tuples) or not. The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples. We illustrate the use of EXISTS—and NOT EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS as in Q16B:

Q16B:	**SELECT**	E.Fname, E.Lname	
	FROM	EMPLOYEE **AS** E	
	WHERE	**EXISTS** (**SELECT**	*
		FROM	DEPENDENT **AS** D
		WHERE	E.Ssn = D.Essn **AND** E.Sex = D.Sex
			AND E.Fname = D.Dependent_name);

EXISTS and NOT EXISTS are typically used in conjunction with a *correlated* nested query. In Q16B, the nested query references the Ssn, Fname, and Sex attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and returns **FALSE** otherwise. On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and returns **FALSE** otherwise. Next, we illustrate the use of NOT EXISTS.

Query 6. Retrieve the names of employees who have no dependents.

Q6:	**SELECT**	Fname, Lname	
	FROM	EMPLOYEE	
	WHERE	**NOT EXISTS** (**SELECT**	*
		FROM	DEPENDENT
		WHERE	Ssn = Essn);

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist,* the EMPLOYEE tuple is selected because the **WHERE**-clause condition will evaluate to **TRUE** in this case. We can explain Q6 as follows: For *each* EMPLOYEE tuple, the correlated nested query selects all

DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

Query 7. List the names of managers who have at least one dependent.

```
Q7:    SELECT     Fname, Lname
       FROM       EMPLOYEE
       WHERE      EXISTS ( SELECT    *
                          FROM      DEPENDENT
                          WHERE     Ssn = Essn )
                  AND
                  EXISTS ( SELECT    *
                          FROM      DEPARTMENT
                          WHERE     Ssn = Mgr_ssn );
```

One way to write this query is shown in Q7, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple. Can you rewrite this query using only a single nested query or no nested queries?

The query Q3: *Retrieve the name of each employee who works on* all *the projects controlled by department number 5* can be written using EXISTS and NOT EXISTS in SQL systems. We show two ways of specifying this query Q3 in SQL as Q3A and Q3B. This is an example of certain types of queries that require *universal quantification*, as we will discuss in Section 8.6.7. One way to write this query is to use the construct (*S2* EXCEPT *S1*) as explained next, and checking whether the result is empty.[1] This option is shown as Q3A.

```
Q3A:   SELECT     Fname, Lname
       FROM       EMPLOYEE
       WHERE      NOT EXISTS ( ( SELECT    Pnumber
                                FROM      PROJECT
                                WHERE     Dnum = 5)
                              EXCEPT    ( SELECT    Pno
                                         FROM      WORKS_ON
                                         WHERE     Ssn = Essn) );
```

In Q3A, the first subquery (which is not correlated with the outer query) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first subquery result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.

---

[1]Recall that EXCEPT is the set difference operator. The keyword MINUS is also sometimes used, for example, in Oracle.

The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3A.

```
Q3B:    SELECT   Lname, Fname
        FROM     EMPLOYEE
        WHERE    NOT EXISTS  ( SELECT    *
                               FROM      WORKS_ON B
                               WHERE   ( B.Pno IN  ( SELECT    Pnumber
                                                     FROM      PROJECT
                                                     WHERE     Dnum = 5 )
                               AND
                               NOT EXISTS  ( SELECT    *
                                             FROM      WORKS_ON C
                                             WHERE     C.Essn = Ssn
                                             AND       C.Pno = B.Pno )));
```

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose Pno is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same Pno and the same Ssn as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. It corresponds to the way we will write this query in tuple relation calculus (see Section 8.6.7).

There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set (no duplicates) or a multiset (duplicates exist).

7.1.5 Explicit Sets and Renaming in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

> **Query 17.** Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:    SELECT   DISTINCT Essn
        FROM     WORKS_ON
        WHERE    Pno IN (1, 2, 3);
```

In SQL, it is possible to **rename** any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names in general, and it can be used in appropriate parts of a query. For example, Q8A shows how query Q8 from Section 4.3.2 can be slightly changed to retrieve the last name of each employee and his or her supervisor while renaming the resulting attribute names

as Employee_name and Supervisor_name. The new names will appear as column headers for the query result.

Q8A:	SELECT	E.Lname **AS** Employee_name, S.Lname **AS** Supervisor_name
	FROM	EMPLOYEE **AS** E, EMPLOYEE **AS** S
	WHERE	E.Super_ssn = S.Ssn;

7.1.6 Joined Tables in SQL and Outer Joins

The concept of a **joined table** (or **joined relation**) was incorporated into SQL to permit users to specify a table resulting from a join operation *in the* FROM *clause* of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause. For example, consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier to specify the join of the EMPLOYEE and DEPARTMENT relations in the WHERE clause, and then to select the desired tuples and attributes. This can be written in SQL as in Q1A:

Q1A:	SELECT	Fname, Lname, Address
	FROM	(EMPLOYEE **JOIN** DEPARTMENT **ON** Dno = Dnumber)
	WHERE	Dname = 'Research';

The FROM clause in Q1A contains a single *joined table.* The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a **NATURAL JOIN** on two relations *R* and *S,* no join condition is specified; an implicit *EQUIJOIN condition* for *each pair of attributes with the same name* from *R* and *S* is created. Each such pair of attributes is included *only once* in the resulting relation (see Sections 8.3.2 and 8.4.4 for more details on the various types of join operations in relational algebra).

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as Dname, Dno (to match the name of the desired join attribute Dno in the EMPLOYEE table), Mssn, and Msdate. The implied join condition for this NATURAL JOIN is EMPLOYEE.Dno = DEPT.Dno, because this is the only pair of attributes with the same name after renaming:

Q1B:	SELECT	Fname, Lname, Address
	FROM	(EMPLOYEE **NATURAL JOIN**
		(DEPARTMENT **AS** DEPT (Dname, Dno, Mssn, Msdate)))
	WHERE	Dname = 'Research';

The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query Q8A, only employees who *have a supervisor* are included in the result;

an EMPLOYEE tuple whose value for Super_ssn is NULL is excluded. If the user requires that all employees be included, a different type of join called **OUTER JOIN** must be used explicitly (see Section 8.4.4 for the definition of OUTER JOIN in relational algebra). There are several variations of OUTER JOIN, as we shall see. In the SQL standard, this is handled by explicitly specifying the keyword OUTER JOIN in a joined table, as illustrated in Q8B:

Q8B:	**SELECT**	E.Lname **AS** Employee_name,
		S.Lname **AS** Supervisor_name
	FROM	(EMPLOYEE **AS** E **LEFT OUTER JOIN** EMPLOYEE **AS** S
		ON E.Super_ssn = S.Ssn);

In SQL, the options available for specifying joined tables include INNER JOIN (only pairs of tuples that match the join condition are retrieved, same as JOIN), LEFT OUTER JOIN (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table), RIGHT OUTER JOIN (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation (see Section 8.2.2), although this should be used only with the utmost care because it generates all possible tuple combinations.

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**. For example, Q2A is a different way of specifying query Q2 from Section 6.3.1 using the concept of a joined table:

Q2A:	**SELECT**	Pnumber, Dnum, Lname, Address, Bdate
	FROM	((PROJECT **JOIN** DEPARTMENT **ON** Dnum = Dnumber)
		JOIN EMPLOYEE **ON** Mgr_ssn = Ssn)
	WHERE	Plocation = 'Stafford';

Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators + =, = +, and + = + for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in Oracle. To specify the left outer join in Q8B using this syntax, we could write the query Q8C as follows:

Q8C:	**SELECT**	E.Lname, S.Lname
	FROM	EMPLOYEE E, EMPLOYEE S
	WHERE	E.Super_ssn + = S.Ssn;

7.1.7 Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. **Grouping** is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database

applications, and we will introduce their use in SQL through examples. A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.[2] The COUNT function returns the *number of tuples or values* as specified in a query. The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.[3] We illustrate the use of these functions with several queries.

> **Query 19.** Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

Q19:	SELECT	**SUM** (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)
	FROM	EMPLOYEE;

This query returns a *single-row* summary of all the rows in the EMPLOYEE table. We could use AS to rename the column names in the resulting single-row table; for example, as in Q19A.

Q19A:	SELECT	**SUM** (Salary) **AS** Total_Sal, **MAX** (Salary) **AS** Highest_Sal,
		MIN (Salary) **AS** Lowest_Sal, **AVG** (Salary) **AS** Average_Sal
	FROM	EMPLOYEE;

If we want to get the preceding aggregate function values for employees of a specific department—say, the 'Research' department—we can write Query 20, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

> **Query 20.** Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

Q20:	SELECT	**SUM** (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)
	FROM	(EMPLOYEE **JOIN** DEPARTMENT **ON** Dno = Dnumber)
	WHERE	Dname = 'Research';

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

Q21:	SELECT	**COUNT** (*)
	FROM	EMPLOYEE;

Q22:	SELECT	COUNT (*)
	FROM	EMPLOYEE, DEPARTMENT
	WHERE	DNO = DNUMBER **AND** DNAME = 'Research';

---

[2]Additional aggregate functions for more advanced statistical calculation were added in SQL-99.

[3]Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, DATE, TIME, and TIMESTAMP domains have total orderings on their values, as do alphabetic strings.

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

Query 23. Count the number of distinct salary values in the database.

Q23:   **SELECT**   **COUNT** (**DISTINCT** Salary)
          FROM   EMPLOYEE;

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute); the only exception is for COUNT(*) because tuples instead of values are counted. In the previous examples, any Salary values that are NULL are not included in the aggregate function calculation. The general rule is as follows: when an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation; if the collection becomes empty because all values are NULL, the aggregate function will return NULL (except in the case of COUNT, where it will return 0 for an empty collection of values).

The preceding examples summarize *a whole relation* (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce a table with a single row or a single value. They illustrate how functions are applied to retrieve a summary value or summary tuple from a table. These functions can also be used in selection conditions involving nested queries. We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents (Query 5), we can write the following:

Q5:   **SELECT**   Lname, Fname
         FROM   EMPLOYEE
         WHERE   (**SELECT**   **COUNT** (*)
                          FROM   DEPENDENT
                          WHERE   Ssn = Essn) >= 2;

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

SQL also has aggregate functions SOME and ALL that can be applied to a collection of Boolean values; SOME returns TRUE if at least one element in the collection is TRUE, whereas ALL returns TRUE if all elements in the collection are TRUE.

7.1.8 Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation,* where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees *in each department* or the number

of employees who work *on each project*. In these cases we need to **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**. We can then apply the function to each such group independently to produce summary information about each group. SQL has a **GROUP BY** clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should *also appear in the SELECT clause,* so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

> **Query 24.** For each department, retrieve the department number, the number of employees in the department, and their average salary.

Q24:	**SELECT**	Dno, **COUNT** (*), **AVG** (Salary)
	FROM	EMPLOYEE
	GROUP BY	Dno;

In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the GROUP BY attribute Dno. Hence, each group contains the employees who work in the same department. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples. Figure 7.1(a) illustrates how grouping works and shows the result of Q24.

If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute.* For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result of Q24.

> **Query 25.** For each project, retrieve the project number, the project name, and the number of employees who work on that project.

Q25:	**SELECT**	Pnumber, Pname, **COUNT** (*)
	FROM	PROJECT, WORKS_ON
	WHERE	Pnumber = Pno
	GROUP BY	Pnumber, Pname;

Q25 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations in the WHERE clause.

Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions.* For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

Figure 7.1
Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

(b)

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

After applying the HAVING clause condition

Query 26. For each project *on which more than two employees work,* retrieve the project number, the project name, and the number of employees who work on the project.

Q26:   **SELECT**   Pnumber, Pname, **COUNT** (*)
          FROM   PROJECT, WORKS_ON
          WHERE   Pnumber = Pno
          GROUP BY   Pnumber, Pname
          HAVING   **COUNT** (*) > 2;

Notice that although selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups.* Figure 7.1(b) illustrates the use of HAVING and displays the result of Q26.

Query 27. For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

Q27:   **SELECT**   Pnumber, Pname, **COUNT** (*)
          FROM   PROJECT, WORKS_ON, EMPLOYEE
          WHERE   Pnumber = Pno **AND** Ssn = Essn **AND** Dno = 5
          GROUP BY   Pnumber, Pname;

In Q27, we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following *incorrect* query:

          SELECT   Dno, **COUNT** (*)
          FROM   EMPLOYEE
          WHERE   Salary>40000
          GROUP BY   Dno
          HAVING   **COUNT** (*) > 5;

This is incorrect because it will select only departments that have more than five employees *who each earn more than $40,000.* The rule is that the WHERE clause is executed first, to select individual tuples or joined tuples; the HAVING clause is applied later, to select individual groups of tuples. In the incorrect query, the tuples are already restricted to employees who earn more than $40,000 *before* the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown in Query 28.

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:    SELECT     Dno, COUNT (*)
        FROM       EMPLOYEE
        WHERE      Salary>40000 AND Dno IN
                   ( SELECT       Dno
                     FROM         EMPLOYEE
        GROUP BY   Dno
                     HAVING       COUNT (*) > 5)
        GROUP BY   Dno;
```

7.1.9 Other SQL Constructs: WITH and CASE

In this section, we illustrate two additional SQL constructs. The WITH clause allows a user to define a table that will only be used in a particular query; it is somewhat similar to creating a view (see Section 7.3) that will be used only in one query and then dropped. This construct was introduced as a convenience in SQL:99 and may not be available in all SQL based DBMSs. Queries using WITH can generally be written using other SQL constructs. For example, we can rewrite Q28 as Q28′:

```
Q28′:   WITH       BIGDEPTS (Dno) AS
                   ( SELECT       Dno
                     FROM         EMPLOYEE
                     GROUP BY     Dno
                     HAVING       COUNT (*) > 5)
        SELECT     Dno, COUNT (*)
        FROM       EMPLOYEE
        WHERE      Salary>40000 AND Dno IN BIGDEPTS
        GROUP BY   Dno;
```

In Q28′, we defined in the WITH clause a temporary table BIG_DEPTS whose result holds the Dno's of departments with more than five employees, then used this table in the subsequent query. Once this query is executed, the temporary table BIGDEPTS is discarded.

SQL also has a CASE construct, which can be used when a value can be different based on certain conditions. This can be used in any part of an SQL query where a value is expected, including when querying, inserting or updating tuples. We illustrate this with an example. Suppose we want to give employees different raise amounts depending on which department they work for; for example, employees in department 5 get a $2,000 raise, those in department 4 get $1,500 and those in department 1 get $3,000 (see Figure 5.6 for the employee tuples). Then we could re-write the update operation U6 from Section 6.4.3 as U6′:

```
U6′:    UPDATE     EMPLOYEE
        SET        Salary =
        CASE       WHEN     Dno = 5     THEN Salary + 2000
                   WHEN     Dno = 4     THEN Salary + 1500
                   WHEN     Dno = 1     THEN Salary + 3000
                   ELSE     Salary + 0 ;
```

In U6′, the salary raise value is determined through the CASE construct based on the department number for which each employee works. The CASE construct can also be used when inserting tuples that can have different attributes being NULL depending on the type of record being inserted into a table, as when a specialization (see Chapter 4) is mapped into a single table (see Chapter 9) or when a union type is mapped into relations.

7.1.10 Recursive Queries in SQL

In this section, we illustrate how to write a recursive query in SQL. This syntax was added in SQL:99 to allow users the capability to specify a recursive query in a declarative manner. An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor. This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of a supervisory employee e at all levels—that is, all employees e' directly supervised by e, all employees e' directly supervised by each employee e', all employees e''' directly supervised by each employee e'', and so on. In SQL:99, this query can be written as follows:

Q29:	**WITH RECURSIVE**	SUP_EMP (SupSsn, EmpSsn) **AS**
	(**SELECT**	SupervisorSsn, Ssn
	FROM	EMPLOYEE
	UNION	
	SELECT	E.Ssn, S.SupSsn
	FROM	EMPLOYEE **AS** E, SUP_EMP **AS** S
	WHERE	E.SupervisorSsn = S.EmpSsn)
	SELECT*	
	FROM	SUP_EMP;

In Q29, we are defining a view SUP_EMP that will hold the result of the recursive query. The view is initially empty. It is first loaded with the first level (supervisor, supervisee) Ssn combinations via the first part (**SELECT** SupervisorSss, Ssn **FROM** EMPLOYEE), which is called the **base query**. This will be combined via UNION with each successive level of supervisees through the second part, where the view contents are joined again with the base values to get the second level combinations, which are UNIONed with the first level. This is repeated with successive levels until a **fixed point** is reached, where no more tuples are added to the view. At this point, the result of the recursive query is in the view SUP_EMP.

7.1.11 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two— SELECT and FROM—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are

specified in the following order, with the clauses between square brackets […] being optional:

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

In order to formulate queries correctly, it is useful to consider the steps that define the *meaning* or *semantics* of each query. A query is evaluated *conceptually*[4] by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause to select and join tuples, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING, and ORDER BY) are specified, we can *think conceptually* of a query as being executed as follows: For *each combination of tuples*—one from each of the relations specified in the FROM clause—evaluate the WHERE clause; if it evaluates to TRUE, place the values of the attributes specified in the SELECT clause from this tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each DBMS has special query optimization routines to decide on an execution plan that is efficient to execute. We discuss query processing and optimization in Chapters 18 and 19.

In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison operator. Some users may be more comfortable with one approach, whereas others may be more comfortable with another. From the programmer's and the system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify

---

[4]The actual order of query evaluation is implementation dependent; this is just a way to conceptually view a query in order to correctly formulate it.

particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient to execute. Ideally, the user should worry only about specifying the query correctly, whereas the DBMS would determine how to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others.

7.2 Specifying Constraints as Assertions and Actions as Triggers

In this section, we introduce two additional features of SQL: the **CREATE ASSERTION** statement and the **CREATE TRIGGER** statement. Section 7.2.1 discusses CREATE ASSERTION, which can be used to specify additional types of constraints that are outside the scope of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity) that we presented in Section 5.2. These built-in constraints can be specified within the **CREATE TABLE** statement of SQL (see Sections 6.1 and 6.2).

In Section 7.2.2 we introduce **CREATE TRIGGER**, which can be used to specify automatic actions that the database system will perform when certain events and conditions occur. This type of functionality is generally referred to as **active databases**. We only introduce the basics of **triggers** in this chapter, and present a more complete discussion of active databases in Section 26.1.

7.2.1 Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints—those that do not fall into any of the categories described in Sections 6.1 and 6.2— via **declarative assertions**, using the **CREATE ASSERTION** statement. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query. For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT      *
                     FROM        EMPLOYEE E, EMPLOYEE M,
                                 DEPARTMENT D
                     WHERE       E.Salary>M.Salary
                        AND      E.Dno = D.Dnumber
                        AND      D.Mgr_ssn = M.Ssn ) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to disable the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions. Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.

The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition.* By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty. In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

Note that the CHECK clause and constraint condition can also be used to specify constraints on *individual* attributes and domains (see Section 6.2.1) and on *individual* tuples (see Section 6.2.4). A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated* in a specific table. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases. The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*. On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.

7.2.2 Introduction to Triggers in SQL

Another important statement in SQL is CREATE TRIGGER. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL. We discuss triggers in detail in Section 26.1 when we describe *active databases*. Here we just give a simple example of how triggers may be used.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database (see Figures 5.5 and 5.6). Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION,[5] which will notify the supervisor. The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

R5: **CREATE TRIGGER** SALARY_VIOLATION
      BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
        ON EMPLOYEE
      FOR EACH ROW
        WHEN (**NEW**.SALARY > (**SELECT** SALARY **FROM** EMPLOYEE
                                WHERE SSN = **NEW**.SUPERVISOR_SSN))
                                INFORM_SUPERVISOR(**NEW**.Supervisor_ssn,
                                NEW.Ssn);

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically. A complete discussion is given in Section 26.1.

---

[5]Assuming that an appropriate external procedure has been declared. We discuss stored procedures in Chapter 10.

7.3 Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We show how views are specified, and then we discuss the problem of updating views and how views can be implemented by the DBMS.

7.3.1 Concept of a View in SQL

A **view** in SQL terminology is a single table that is derived from other tables.[6] These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically. For example, referring to the COMPANY database in Figure 5.5, we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE, WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single-table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS_ON, and PROJECT tables the **defining tables** of the view.

7.3.2 Specification of Views in SQL

In SQL, the command to specify a view is **CREATE VIEW**. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 7.2 when applied to the database schema of Figure 5.5.

V1:	**CREATE VIEW**	WORKS_ON1
	AS SELECT	Fname, Lname, Pname, Hours
	FROM	EMPLOYEE, PROJECT, WORKS_ON
	WHERE	Ssn = Essn **AND** Pno = Pnumber;
V2:	**CREATE VIEW**	DEPT_INFO(Dept_name, No_of_emps, Total_sal)
	AS SELECT	Dname, **COUNT** (*), **SUM** (Salary)
	FROM	DEPARTMENT, EMPLOYEE
	WHERE	Dnumber = Dno
	GROUP BY	Dname;

---

[6]As used in SQL, the term *view* is more limited than the term *user view* discussed in Chapters 1 and 2, since a user view would possibly include many relations.

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

Figure 7.2

Two views specified on the database schema of Figure 5.5.

In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON. View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as in QV1:

> QV1: **SELECT** Fname, Lname
> **FROM** WORKS_ON1
> **WHERE** Pname = 'ProductX';

The same query would require the specification of two joins if specified on the base relations directly; one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism (see Section 7.3.4 and Chapter 30).

A view is supposed to be *always up-to-date*; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view does not have to be realized or materialized at the time of *view definition* but rather at the time when we *specify a query* on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date. We will discuss various ways the DBMS can utilize to keep a view up-to-date in the next subsection.

If we do not need a view anymore, we can use the **DROP VIEW** command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement in V1A:

> V1A: **DROP VIEW** WORKS_ON1;

7.3.3 View Implementation, View Update, and Inline Views

The problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the

user) into a query on the underlying base tables. For example, the query QV1 would be automatically modified to the following query by the DBMS:

```
SELECT     Fname, Lname
FROM       EMPLOYEE, PROJECT, WORKS_ON
WHERE      Ssn = Essn AND Pno = Pnumber
           AND Pname = 'ProductX';
```

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple view queries are going to be applied to the same view within a short period of time. The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied *to one of the defining base tables*. The view is generally kept as a materialized (physically stored) table as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.

Different strategies as to when a materialized view is updated are possible. The **immediate update** strategy updates a view as soon as the base tables are changed; the **lazy update** strategy updates the view when needed by a view query; and the **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

A user can always issue a retrieval query against any view. However, issuing an INSERT, DELETE, or UPDATE command on a view table is in many cases not possible. In general, an update on a view defined on a *single table* without any *aggregate functions* can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in *multiple ways*. Hence, it is often not possible for the DBMS to determine which of the updates is intended. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1:    UPDATE WORKS_ON1
        SET        Pname = 'ProductY'
        WHERE      Lname = 'Smith' AND Fname = 'John'
                   AND Pname = 'ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create

additional side effects that affect the result of other queries. For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

(a):   **UPDATE** WORKS_ON
        SET   Pno =   (**SELECT** Pnumber
                                  FROM   PROJECT
                                  WHERE   Pname = 'ProductY')
        WHERE   Essn **IN**   (**SELECT**   Ssn
                                    FROM   EMPLOYEE
                                  WHERE   Lname = 'Smith' **AND** Fname = 'John')
               AND
               Pno =   (**SELECT**   Pnumber
                                    FROM   PROJECT
                                  WHERE   Pname = 'ProductX');

(b):   **UPDATE** PROJECT   **SET**   Pname = 'ProductY'
        WHERE   Pname = 'ProductX';

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view does not make sense because Total_sal is defined to be the sum of the individual employee salaries. This incorrect request is shown as UV2:

UV2:   **UPDATE**   DEPT_INFO
        SET   Total_sal = 100000
        WHERE   Dname = 'Research';

Generally, a view update is feasible when only *one possible update* on the base relations can accomplish the desired update operation on the view. Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, it is usually not permitted. Some researchers have suggested that the DBMS have a certain procedure for choosing one of the possible updates as the most likely one. Some researchers have developed methods for choosing the most likely update, whereas other researchers prefer to have the user choose the desired update mapping during view definition. But these options are generally not available in most commercial DBMSs.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have* default values specified.

- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause **WITH CHECK OPTION** should be added at the end of the view definition if a view *is to be updated* by INSERT, DELETE, or UPDATE statements. This allows the system to reject operations that violate the SQL rules for view updates. The full set of SQL rules for when a view may be modified by the user are more complex than the rules stated earlier.

It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

7.3.4 Views as Authorization Mechanisms

We describe SQL query authorization statements (GRANT and REVOKE) in detail in Chapter 30, when we present database security and authorization mechanisms. Here, we will just give a couple of simple examples to illustrate how views can be used to hide certain attributes or tuples from unauthorized users. Suppose a certain user is only allowed to see employee information for employees who work for department 5; then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself. This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW      DEPT5EMP      AS
SELECT           *
FROM             EMPLOYEE
WHERE            Dno = 5;
```

In a similar manner, a view can restrict a user to only see certain columns; for example, only the first name, last name, and address of an employee may be visible as follows:

```
CREATE VIEW      BASIC_EMP_DATA      AS
SELECT           Fname, Lname, Address
FROM             EMPLOYEE;
```

Thus by creating an appropriate view and granting certain users access to the view and not the base tables, they would be restricted to retrieving only the data specified in the view. Chapter 30 discusses security and authorization in detail, including the GRANT and REVOKE statements of SQL.

7.4 Schema Change Statements in SQL

In this section, we give an overview of the **schema evolution commands** available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema. Certain

checks must be done by the DBMS to ensure that the changes do not affect the rest of the database and make it inconsistent.

7.4.1 The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, types, or constraints. One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY **CASCADE**;

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 6.1, we can get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT **CASCADE**;

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views (see Section 7.3) or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command (see Section 6.4.2) should be used instead of DROP TABLE.

The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

7.4.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible **alter table actions** include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema (see Figure 6.1), we can use the command

ALTER TABLE COMPANY.EMPLOYEE **ADD COLUMN** Job VARCHAR(12);

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple (see Section 6.4.3). If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other schema elements) reference the column. For example, the following command removes the attribute Address from the EMPLOYEE base table:

> **ALTER TABLE** COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

> **ALTER TABLE** COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn
> **DROP DEFAULT**;
> **ALTER TABLE** COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn
> **SET DEFAULT** '333445555';

One can also change the constraints specified on a table by adding or dropping a named constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 6.2 from the EMPLOYEE relation, we write:

> **ALTER TABLE** COMPANY.EMPLOYEE
> **DROP CONSTRAINT** EMPSUPERFK **CASCADE**;

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the **ADD CONSTRAINT** keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.

The preceding subsections gave an overview of the schema evolution commands of SQL. It is also possible to create new tables and views within a database schema using the appropriate commands. There are many other details and options; we refer the interested reader to the SQL documents listed in the Selected Bibliography at the end of this chapter.

7.5 Summary

In this chapter we presented additional features of the SQL database language. We started in Section 7.1 by presenting more complex features of SQL retrieval queries, including nested queries, joined tables, outer joins, aggregate functions, and grouping. In Section 7.2, we described the CREATE ASSERTION statement, which allows the specification of more general constraints on the database, and introduced the

concept of triggers and the CREATE TRIGGER statement. Then, in Section 7.3, we described the SQL facility for defining views on the database. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables. Section 7.4 introduced the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints.

Table 7.2 summarizes the syntax (or structure) of various SQL statements. This summary is not meant to be comprehensive or to describe every possible SQL construct; rather, it is meant to serve as a quick reference to the major types of constructs available in SQL. We use BNF notation, where nonterminal symbols

Table 7.2   Summary of SQL Syntax

CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
                          { , <column name> <column type> [<attribute constraint>] }
                          [<table constraint> { , <table constraint> }])

DROP TABLE <table name>   ALTER TABLE <table name> ADD <column name> <column type>
SELECT [DISTINCT] <attribute list>   FROM (<table name> { <alias> } | <joined table>) { , (<table name> { <alias> } | <joined table>) }   [WHERE <condition>]   [GROUP BY <grouping attributes> [HAVING <group selection condition>]]   [ORDER BY <column name> [<order>] { , <column name> [<order>] }]   <attribute list> ::= (* | (<column name> | <function> (([DISTINCT] <column name> | *)))   { , (<column name> | <function> (([DISTINCT] <column name> | *)) }))   <grouping attributes> ::= <column name> { , <column name> }
<order> ::= (ASC | DESC)
INSERT INTO <table name> [(<column name> { , <column name> })]   (VALUES (<constant value> , { <constant value> }) { , (<constant value> { , <constant value> }) } | <select statement>)
DELETE FROM <table name>   [WHERE <selection condition>]
UPDATE <table name>   SET <column name> = <value expression> { , <column name> = <value expression> }   [WHERE <selection condition>]
CREATE [UNIQUE] INDEX <index name>   ON <table name> (<column name> [<order>] { , <column name> [<order>] })   [CLUSTER]
DROP INDEX <index name>
CREATE VIEW <view name> [(<column name> { , <column name> })]   AS <select statement>
DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

are shown in angled brackets < … >, optional parts are shown in square brackets […], repetitions are shown in braces { … }, and alternatives are shown in parentheses (… | … | …).[7]

Review Questions

7.1. Describe the six clauses in the syntax of an SQL retrieval query. Show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?

7.2. Describe conceptually how an SQL retrieval query will be executed by specifying the conceptual order of executing each of the six clauses.

7.3. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?

7.4. Discuss how each of the following constructs is used in SQL, and discuss the various options for each construct. Specify what each construct is useful for.
   a. Nested queries
   b. Joined tables and outer joins
   c. Aggregate functions and grouping
   d. Triggers
   e. Assertions and how they differ from triggers
   f. The SQL WITH clause
   g. SQL CASE construct
   h. Views and their updatability
   i. Schema change commands

Exercises

7.5. Specify the following queries on the database in Figure 5.5 in SQL. Show the query results if each query is applied to the database state in Figure 5.6.
   a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.
   b. Suppose that we want the number of *male* employees in each department making more than $30,000, rather than all employees (as in Exercise 7.5a). Can we specify this query in SQL? Why or why not?

---

[7]The full syntax of SQL is described in many voluminous documents of hundreds of pages.

7.6. Specify the following queries in SQL on the database schema in Figure 1.2.

a. Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).

b. Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.

7.7. In SQL, specify the following queries on the database in Figure 5.5 using the concept of nested queries and other concepts described in this chapter.

a. Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.

b. Retrieve the names of all employees whose supervisor's supervisor has '888665555' for Ssn.

c. Retrieve the names of employees who make at least $10,000 more than the employee who is paid the least in the company.

7.8. Specify the following views in SQL on the COMPANY database schema shown in Figure 5.5.

a. A view that has the department name, manager name, and manager salary for every department

b. A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department

c. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project

d. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project *with more than one employee working on it*

7.9. Consider the following view, DEPT_SUMMARY, defined on the COMPANY database in Figure 5.6:

```
CREATE VIEW      DEPT_SUMMARY (D, C, Total_s, Average_s)
AS SELECT        Dno, COUNT (*), SUM (Salary), AVG (Salary)
FROM             EMPLOYEE
GROUP BY         Dno;
```

State which of the following queries and updates would be allowed on the view. If a query or update would be allowed, show what the corresponding query or update on the base relations would look like, and give its result when applied to the database in Figure 5.6.

a. ```
   SELECT   *
   FROM   DEPT_SUMMARY;
   ```

b. ```
   SELECT      D, C
   FROM        DEPT_SUMMARY
   WHERE       TOTAL_S > 100000;
   ```

    c. **SELECT**   D, AVERAGE_S
       FROM   DEPT_SUMMARY
       WHERE   C > (**SELECT** C **FROM** DEPT_SUMMARY **WHERE** D = 4);

    d. **UPDATE**   DEPT_SUMMARY
       SET   D = 3
       WHERE   D = 4;

    e. **DELETE**   **FROM** DEPT_SUMMARY
       WHERE   C > 4;

Selected Bibliography

Reisner (1977) describes a human factors evaluation of SEQUEL, a precursor of SQL, in which she found that users have some difficulty with specifying join conditions and grouping correctly. Date (1984) contains a critique of the SQL language that points out its strengths and shortcomings. Date and Darwen (1993) describes SQL2. ANSI (1986) outlines the original SQL standard. Various vendor manuals describe the characteristics of SQL as implemented on DB2, SQL/DS, Oracle, INGRES, Informix, and other commercial DBMS products. Melton and Simon (1993) give a comprehensive treatment of the ANSI 1992 standard called SQL2. Horowitz (1992) discusses some of the problems related to referential integrity and propagation of updates in SQL2.

The question of view updates is addressed by Dayal and Bernstein (1978), Keller (1982), and Langerak (1990), among others. View implementation is discussed in Blakeley et al. (1989). Negri et al. (1991) describes formal semantics of SQL queries.

There are many books that describe various aspects of SQL. For example, two references that describe SQL-99 are Melton and Simon (2002) and Melton (2003). Further SQL standards—SQL 2006 and SQL 2008—are described in a variety of technical reports; but no standard references exist.

The Relational Algebra and Relational Calculus

I n this chapter we discuss the two *formal languages* for the relational model: the relational algebra and the relational calculus. In contrast, Chapters 6 and 7 described the *practical language* for the relational model, namely the SQL standard. Historically, the relational algebra and calculus were developed before the SQL language. SQL is primarily based on concepts from relational calculus and has been extended to incorporate some concepts from relational algebra as well. Because most relational DBMSs use SQL as their language, we presented the SQL language first.

Recall from Chapter 2 that a data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining the database's structure and constraints. We presented the structures and constraints of the formal relational model in Chapter 5. The basic set of operations for the formal relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as *relational algebra expressions*. The result of a retrieval query is a new relation. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs), as we shall discuss in Chapters 18 and 19. Third, some of its concepts are incorporated into the SQL standard

239

query language for RDBMSs. Although most commercial RDBMSs in use today do not provide user interfaces for relational algebra queries, the core operations and functions in the internal modules of most relational systems are based on relational algebra operations. We will define these operations in detail in Sections 8.1 through 8.4 of this chapter.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in a variation of relational calculus known as the tuple relational calculus.[1]

The relational algebra is often considered to be an integral part of the relational data model. Its operations can be divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the *formal* relational model (see Section 5.1). Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT (also known as CROSS PRODUCT). The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. First, we describe the SELECT and PROJECT operations in Section 8.1 because they are **unary operations** that operate on single relations. Then we discuss set operations in Section 8.2. In Section 8.3, we discuss JOIN and other complex **binary operations**, which operate on two tables by combining related tuples (records) based on *join conditions*. The COMPANY relational database shown in Figure 5.6 is used for our examples.

Some common database requests cannot be performed with the original relational algebra operations, so additional operations were created to express these requests. These include **aggregate functions**, which are operations that can *summarize* data from the tables, as well as additional types of JOIN and UNION operations, known as OUTER JOINs and OUTER UNIONs. These operations, which were added to the original relational algebra because of their importance to many database applications, are described in Section 8.4. We give examples of specifying queries that use relational operations in Section 8.5. Some of these same queries were used in Chapters 6 and 7. By using the same query numbers in this chapter, the reader can contrast how the same queries are written in the various query languages.

In Sections 8.6 and 8.7 we describe the other main formal language for relational databases, the **relational calculus**. There are two variations of relational calculus. The *tuple* relational calculus is described in Section 8.6 and the *domain* relational calculus is described in Section 8.7. Some of the SQL constructs discussed in

---

[1]SQL is based on tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions, as illustrated in Chapters 6, 7, and 9.

Chapters 6 and 7 are based on the tuple relational calculus. The relational calculus is a formal language, based on the branch of mathematical logic called predicate calculus.[2] In tuple relational calculus, variables range over *tuples*, whereas in domain relational calculus, variables range over the *domains* (values) of attributes. In Appendix C we give an overview of the Query-By-Example (QBE) language, which is a graphical user-friendly relational language based on domain relational calculus. Section 8.8 summarizes the chapter.

For the reader who is interested in a less detailed introduction to formal relational languages, Sections 8.4, 8.6, and 8.7 may be skipped.

8.1 Unary Relational Operations: SELECT and PROJECT

8.1.1 The SELECT Operation

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**.[3] We can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. Alternatively, we can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition. The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are filtered out. For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$$\sigma_{Dno=4}(\text{EMPLOYEE})$$
$$\sigma_{Salary>30000}(\text{EMPLOYEE})$$

In general, the SELECT operation is denoted by

$$\sigma_{<\text{selection condition}>}(R)$$

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation R. Notice that R is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as R.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>

---

[2]In this chapter no familiarity with first-order predicate calculus—which deals with quantified variables and values—is assumed.

[3]The SELECT operation is **different** from the SELECT clause of SQL. The SELECT operation chooses tuples from a table, and is sometimes called a RESTRICT or FILTER operation.

or

<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of R, <comparison op> is nor-
mally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant
value from the attribute domain. Clauses can be connected by the standard Boolean
operators *and*, *or*, and *not* to form a general selection condition. For example, to
select the tuples for all employees who either work in department 4 and make over
$25,000 per year, or work in department 5 and make over $30,000, we can specify
the following SELECT operation:

$$\sigma_{(\text{Dno}=4 \text{ AND } \text{Salary}>25000) \text{ OR } (\text{Dno}=5 \text{ AND } \text{Salary}>30000)}(\text{EMPLOYEE})$$

The result is shown in Figure 8.1(a).

Notice that all the comparison operators in the set {=, <, ≤, >, ≥, ≠} can apply to
attributes whose domains are *ordered values*, such as numeric or date domains.
Domains of strings of characters are also considered to be ordered based on the col-
lating sequence of the characters. If the domain of an attribute is a set of *unordered
values*, then only the comparison operators in the set {=, ≠} can be used. An exam-
ple of an unordered domain is the domain Color = { 'red', 'blue', 'green', 'white',
'yellow', …}, where no order is specified among the various colors. Some domains
allow additional types of comparison operators; for example, a domain of character
strings may allow the comparison operator SUBSTRING_OF.

Figure 8.1
Results of SELECT and PROJECT operations. (a) $\sigma_{(\text{Dno}=4 \text{ AND } \text{Salary}>25000) \text{ OR } (\text{Dno}=5 \text{ AND } \text{Salary}>30000)}$ (EMPLOYEE).
(b) $\pi_{\text{Lname, Fname, Salary}}$(EMPLOYEE). (c) $\pi_{\text{Sex, Salary}}$(EMPLOYEE).

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

In general, the result of a SELECT operation can be determined as follows. The <selection condition> is applied independently to each *individual tuple t* in *R*. This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to TRUE, then tuple *t* is **selected**. All the selected tuples appear in the result of the SELECT operation. The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (**NOT** cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is **unary**; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple. The **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of *R*. The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in *R*. That is, $|\sigma c\,(R)| \leq |R|$ for any condition *C*. The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(...\,(\sigma_{<condn>}(R))\,...)) = \sigma_{<cond1>\,\mathbf{AND}<cond2>\,\mathbf{AND}...\mathbf{AND}\,<condn>}(R)$$

In SQL, the SELECT condition is typically specified in the *WHERE clause* of a query. For example, the following operation:

$$\sigma_{Dno=4\ \mathbf{AND}\ Salary>25000}\ (EMPLOYEE)$$

would correspond to the following SQL query:

```
SELECT    *
FROM      EMPLOYEE
WHERE     Dno=4 AND Salary>25000;
```

8.1.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. Therefore, the result of the PROJECT operation can be visualized as a *vertical partition* of the relation into two relations:

one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$$

The resulting relation is shown in Figure 8.1(b). The general form of the PROJECT operation is

$$\pi_{<\text{attribute list}>}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation R. Again, notice that R is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.

If the attribute list includes only nonkey attributes of R, duplicate tuples are likely to occur. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**. For example, consider the following PROJECT operation:

$$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

The result is shown in Figure 8.1(c). Notice that the tuple <'F', 25000> appears only once in Figure 8.1(c), even though this combination of values appears twice in the EMPLOYEE relation. Duplicate elimination involves sorting or some other technique to detect duplicates and thus adds more processing. If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. This was not permitted in the formal relational model but is allowed in SQL (see Section 6.3).

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R. If the projection list is a superkey of R—that is, it includes some key of R—the resulting relation has the *same number* of tuples as R. Moreover,

$$\pi_{<\text{list1}>}(\pi_{<\text{list2}>}(R)) = \pi_{<\text{list1}>}(R)$$

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

In SQL, the PROJECT attribute list is specified in the *SELECT clause* of a query. For example, the following operation:

$$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT      DISTINCT Sex, Salary
FROM        EMPLOYEE
```

Notice that if we remove the keyword **DISTINCT** from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra, but the algebra can be extended to include this operation and allow relations to be multisets; we do not discuss these extensions here.

8.1.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 8.1 that depict operation results do not have any names. In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results. For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

Figure 8.2(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, and using the **assignment operation,** denoted by $\leftarrow$ (left arrow), as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$$

It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to **rename** the attributes in the intermediate and result relations. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$\text{TEMP} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLOYEE})$$
$$R(\text{First\_name, Last\_name, Salary}) \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{TEMP})$$

These two operations are illustrated in Figure 8.2(b).

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

$$\rho_{S(B1, B2, \dots, Bn)}(R) \quad \text{or} \quad \rho_S(R) \quad \text{or} \quad \rho_{(B1, B2, \dots, Bn)}(R)$$

(a)

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston,TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble,TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figure 8.2
Results of a sequence of operations. (a) $\pi_{\text{Fname, Lname, Salary}} (\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$.
(b) Using intermediate relations and renaming of attributes.

---

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new rela-
tion name, and B_1, B_2, ... , B_n are the new attribute names. The first expression
renames both the relation and its attributes, the second renames the relation only,
and the third renames the attributes only. If the attributes of R are $(A_1, A_2, \ldots , A_n)$
in that order, then each A_i is renamed as B_i.

In SQL, a single query typically represents a complex relational algebra expression.
Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

> **SELECT**   E.Fname **AS** First_name, E.Lname **AS** Last_name, E.Salary **AS** Salary
> **FROM**   EMPLOYEE **AS** E
> **WHERE**   E.Dno=5,

8.2 Relational Algebra Operations from Set Theory

8.2.1 The UNION, INTERSECTION, and MINUS Operations

The next group of relational algebra operations are the standard mathematical
operations on sets. For example, to retrieve the Social Security numbers of all

employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:[4]

DEP5_EMPS ← $\sigma_{Dno=5}$(EMPLOYEE)
RESULT1 ← π_{Ssn}(DEP5_EMPS)
RESULT2(Ssn) ← π_{Super_ssn}(DEP5_EMPS)
RESULT ← RESULT1 ∪ RESULT2

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 8.3) while eliminating any duplicates. Thus, the Ssn value '333445555' appears only once in the result.

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**). These are **binary** operations; that is, each is applied to two sets (of tuples). When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility* or *type compatibility*. Two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ are said to be **union compatible** (or **type compatible**) if they have the same degree n and if $dom(A_i) = dom(B_i)$ for $1 \le i \le n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

- UNION: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- INTERSECTION: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
- SET DIFFERENCE (or MINUS): The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

Figure 8.3
Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2.

[4]As a single relational algebra expression, this becomes Result ← π_{Ssn} ($\sigma_{Dno=5}$ (EMPLOYEE)) ∪ π_{Super_ssn} ($\sigma_{Dno=5}$ (EMPLOYEE)).

We will adopt the convention that the resulting relation has the same attribute names as the *first* relation R. It is always possible to rename the attributes in the result using the rename operator.

Figure 8.4 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 8.4(a) are union compatible and their tuples represent the names of students and the names of instructors, respectively. The result of the UNION operation in Figure 8.4(b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 8.4(c)) includes only those who are both students and instructors.

Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R \quad \text{and} \quad R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as *n-ary* operations applicable to any number of relations because both are also *associative operations;* that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \quad \text{and} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

Figure 8.4
The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations.
(b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT − INSTRUCTOR.
(e) INSTRUCTOR − STUDENT.

(a) STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

The MINUS operation is *not commutative;* that is, in general,

$R - S \neq S - R$

Figure 8.4(d) shows the names of students who are not instructors, and Figure 8.4(e) shows the names of instructors who are not students.

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

$R \cap S = ((R \cup S) - (R - S)) - (S - R)$

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations described here. In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates (see Section 6.3.4).

8.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

Next, we discuss the **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**−which is denoted by ×. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

The *n*-ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from *n* underlying relations. The CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

> FEMALE_EMPS ← $\sigma_{Sex='F'}$(EMPLOYEE)
> EMPNAMES ← $\pi_{Fname, Lname, Ssn}$(FEMALE_EMPS)
> EMP_DEPENDENTS ← EMPNAMES × DEPENDENT
> ACTUAL_DEPENDENTS ← $\sigma_{Ssn=Essn}$(EMP_DEPENDENTS)
> RESULT ← $\pi_{Fname, Lname, Dependent\_name}$(ACTUAL_DEPENDENTS)

The resulting relations from this sequence of operations are shown in Figure 8.5. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 8.5 with DEPENDENT from Figure 5.6. In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful (every dependent is

Figure 8.5
The CARTESIAN PRODUCT (CROSS PRODUCT) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

combined with *every* female employee). We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn value match the Ssn value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all. It is the responsibility of the user to make sure to apply only meaningful operations to relations.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *related tuples* from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation. We discuss the JOIN operation next.

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables (see Section 7.1.6). Alternatively, if there are two tables in the FROM clause and there is no corresponding join condition in the WHERE clause of the SQL query, the result will also be the CARTESIAN PRODUCT of the two tables (see Q10 in Section 6.3.3).

8.3 Binary Relational Operations: JOIN and DIVISION

8.3.1 The JOIN Operation

The **JOIN** operation, denoted by ⋈, is used to combine *related tuples* from two relations into single "longer" tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations. To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple. We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

DEPT_MGR ← DEPARTMENT ⋈ $_{Mgr\_ssn=Ssn}$ EMPLOYEE
RESULT ← $\pi_{Dname, Lname, Fname}$(DEPT_MGR)

The first operation is illustrated in Figure 8.6. Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used frequently when specifying database queries. Consider the earlier example

Figure 8.6

Result of the JOIN operation DEPT_MGR ← DEPARTMENT ⋈ $_{Mgr\_ssn=Ssn}$EMPLOYEE.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	· · ·	Fname	Minit	Lname	Ssn	· · ·
Research	5	333445555	· · ·	Franklin	T	Wong	333445555	· · ·
Administration	4	987654321	· · ·	Jennifer	S	Wallace	987654321	· · ·
Headquarters	1	888665555	· · ·	James	E	Borg	888665555	· · ·

illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

> EMP_DEPENDENTS ← EMPNAMES × DEPENDENT
> ACTUAL_DEPENDENTS ← $\sigma_{Ssn=Essn}$(EMP_DEPENDENTS)

These two operations can be replaced with a single JOIN operation as follows:

> ACTUAL_DEPENDENTS ← EMPNAMES ⋈ $_{Ssn=Essn}$DEPENDENT

The general form of a JOIN operation on two relations[5] $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is

> $R ⋈_{<join\ condition>}S$

The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition*. This is the main difference between CARTESIAN PRODUCT and JOIN. In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q *as a single combined tuple*.

A general join condition is of the form

> <condition> **AND** <condition> **AND ... AND** <condition>

where each <condition> is of the form $A_i \theta B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. A JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

---

[5]Again, notice that R and S can be any relations that result from general *relational algebra expressions*.

8.3.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. Both previous examples were EQUIJOINs. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple. For example, in Figure 8.6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.[6] The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

> PROJ_DEPT ← PROJECT * $\rho_{(Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)}$(DEPARTMENT)

The same query can be done in two steps by creating an intermediate table DEPT as follows:

> DEPT ← $\rho_{(Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)}$(DEPARTMENT)
> PROJ_DEPT ← PROJECT * DEPT

The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 8.7(a). In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but *only one join attribute value* is kept.

If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

> DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS

The resulting relation is shown in Figure 8.7(b), which combines each department with its locations and has one tuple for each location. In general, the join condition for NATURAL JOIN is constructed by equating *each pair of join attributes* that have the same name in the two relations and combining these conditions with **AND**. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

---

[6]NATURAL JOIN is basically an EQUIJOIN followed by the removal of the superfluous attributes.

(a)

PROJ_DEPT

Pname	Pnumber	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 8.7
Results of two natural join operations. (a) proj_dept ← project * dept.
(b) dept_locs ← department * dept_locations.

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie _{<join\ condition>} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity**, which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

As we can see, a single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called *outer joins* (see Section 8.4.4). Informally, an *inner join* is a type of match-and-combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. Note that sometimes a join may be specified between a relation and itself, as we will illustrate in Section 8.4.3. The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

$$((PROJECT \bowtie _{Dnum=Dnumber}DEPARTMENT) \bowtie _{Mgr\_ssn=Ssn}EMPLOYEE)$$

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

In SQL, JOIN can be realized in several different ways. The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions. This is very common and is illustrated by queries Q1, Q1A, Q1B, Q2, and Q8 in Sections 6.3.1 and 6.3.2, as well as by many other query examples in Chapters 6 and 7. The second way is to use a nested relation, as illustrated by queries Q4A and Q16 in Section 7.1.2. Another way is to use the concept of joined tables, as illustrated by the queries Q1A, Q1B, Q8B, and Q2A in Section 7.1.6. The construct of joined tables was added to SQL2 to allow the user to specify explicitly all the various types of joins, because the other methods were more limited. It also allows the user to clearly distinguish join conditions from the selection conditions in the WHERE clause.

8.3.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ is a **complete** set; that is, any of the other original relational algebra operations can be expressed as a *sequence of operations from this set*. For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \bowtie_{<condition>} S \equiv \sigma_{<condition>}(R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra. However, they are important to include as separate operations because they are convenient to use and are very commonly applied in database applications. Other operations have been included in the basic relational algebra for convenience rather than necessity. We discuss one of these—the DIVISION operation—in the next section.

8.3.4 The DIVISION Operation

The DIVISION operation, denoted by $\div$, is useful for a special kind of query that sometimes occurs in database applications. An example is *Retrieve the names of employees who work on **all** the projects that 'John Smith' works on*. To express this query using the DIVISION operation, proceed as follows. First, retrieve the

list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

SMITH ← $\sigma_{Fname='John'\ \textbf{AND}\ Lname='Smith'}$(EMPLOYEE)
SMITH_PNOS ← π_{Pno}(WORKS_ON ⋈ $_{Essn=Ssn}$SMITH)

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

SSN_PNOS ← $\pi_{Essn,\ Pno}$(WORKS_ON)

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

SSNS(Ssn) ← SSN_PNOS ÷ SMITH_PNOS
RESULT ← $\pi_{Fname,\ Lname}$(SSNS * EMPLOYEE)

The preceding operations are shown in Figure 8.8(a).

In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where the attributes of S are a subset of the attributes of R; that is, $X \subseteq Z$. Let Y be the set of attributes of R that are not attributes of S; that is, $Y = Z - X$ (and hence $Z = X \cup Y$).

Figure 8.8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R [Y] = t$, and with $t_R [X] = t_S$ for *every* tuple t_S in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every tuple* in S. Note that in the formulation of the DIVISION operation, the tuples in the denominator relation S restrict the numerator relation R by selecting those tuples in the result that match all values present in the denominator. It is not necessary to know what those values are as they can be computed by another operation, as illustrated in the SMITH_PNOS relation in the previous example.

Figure 8.8(b) illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$. Notice that the tuples (values) b_1 and b_4 appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: b_2 does not appear with a_2, and b_3 does not appear with a_1.

The DIVISION operation can be expressed as a sequence of π, $\times$, and $-$ operations as follows:

$$T1 \leftarrow \pi_Y(R)$$
$$T2 \leftarrow \pi_Y((S \times T1) - R)$$
$$T \leftarrow T1 - T2$$

The DIVISION operation is defined for convenience for dealing with queries that involve *universal quantification* (see Section 8.6.7) or the *all* condition. Most RDBMS implementations with SQL as the primary query language do not directly implement division. SQL has a roundabout way of dealing with the type of query just illustrated (see Section 7.1.4, queries Q3A and Q3B). Table 8.1 lists the various basic relational algebra operations we have discussed.

8.3.5 Notation for Query Trees

In this section we describe a notation typically used in relational DBMSs (RDBMSs) to represent queries internally. The notation is called a *query tree* or sometimes it is known as a *query evaluation tree* or *query execution tree*. It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Figure 8.9 shows a query tree for Query 2 (see Section 6.3.1): *For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.* This query is specified

Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{<\text{join condition}>} R_2$, OR $R_1 \bowtie_{(<\text{join attributes 1}>)}$, $_{(<\text{join attributes 2}>)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{<\text{join condition}>} R_2$, OR $R_1 *_{(<\text{join attributes 1}>)}$, $_{(<\text{join attributes 2}>)}$ R_2 OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

on the relational schema of Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT}))$$
$$\bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

In Figure 8.9, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE. The relational algebra operations in the expression are represented by internal tree nodes. The query tree signifies an explicit order of execution in the following sense. In order to execute Q2, the node marked (1) in Figure 8.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly,

Figure 8.9
Query tree corresponding to the relational algebra expression for Q2.

node (2) must begin to execute and produce results before node (3) can start execution, and so on. In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra. We will revisit query trees when we discuss query processing and optimization in Chapters 18 and 19.

8.4 Additional Relational Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations described in Sections 8.1 through 8.3. In this section we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

8.4.1 Generalized Projection

The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list. The generalized form can be expressed as:

$$\pi_{F1, \, F2, \, ..., \, Fn} (R)$$

where $F_1, F_2, \ldots, F_n$ are functions over the attributes in relation R and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result.

As an example, consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary − Deduction,
Bonus = 2000 ∗ Years_service, and
Tax = 0.25 ∗ Salary

Then a generalized projection combined with renaming may be used as follows:

REPORT ← $\rho_{(Ssn,\ Net\_salary,\ Bonus,\ Tax)}(\pi_{Ssn,\ Salary\ -\ Deduction,\ 2000\ *\ Years\_service,\ 0.25\ *\ Salary}$(EMPLOYEE))

8.4.2 Aggregate Functions and Grouping

Another type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are used in simple statistical queries that summarize information from the database tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function *independently to each group*. An example would be to group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department. We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.

We can define an AGGREGATE FUNCTION operation, using the symbol I (pronounced *script F*)[7], to specify these types of requests as follows:

$_{<grouping\ attributes>}\ \Im\ _{<function\ list>}\ (R)$

where <grouping attributes> is a list of attributes of the relation specified in *R*, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and <attribute> is an attribute of the relation specified by *R*. The resulting relation has the grouping attributes plus one attribute for each element in the function list. For example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

$\rho_{R(Dno,\ No\_of\_employees,\ Average\_sal)}\ (_{Dno}\ \Im\ _{COUNT\ Ssn,\ AVERAGE\ Salary}$ (EMPLOYEE))

---

[7]There is no single agreed-upon notation for specifying aggregate functions. In some cases a "script A" is used.

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

Figure 8.10
The aggregate function operation.

a. $\rho_{R(Dno,\ No\_of\_employees,\ Average\_sal)}(_{Dno}\,\Im\,_{COUNT\ Ssn,\ AVERAGE\ Salary}\,(EMPLOYEE))$.

b. $_{Dno}\,\Im\,_{COUNT\ Ssn,\ AVERAGE\ Salary}(EMPLOYEE)$.

c. $\Im\,_{COUNT\ Ssn,\ AVERAGE\ Salary}(EMPLOYEE)$.

The result of this operation on the EMPLOYEE relation of Figure 5.6 is shown in Figure 8.10(a).

In the preceding example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation R. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>.[8] For example, Figure 8.10(b) shows the result of the following operation:

$$_{Dno}\,\Im\,_{COUNT\ Ssn,\ AVERAGE\ Salary}(EMPLOYEE)$$

If no grouping attributes are specified, the functions are applied to *all the tuples* in the relation, so the resulting relation has a *single tuple only*. For example, Figure 8.10(c) shows the result of the following operation:

$$\Im\,_{COUNT\ Ssn,\ AVERAGE\ Salary}(EMPLOYEE)$$

It is important to note that, in general, duplicates are *not eliminated* when an aggregate function is applied; this way, the normal interpretation of functions such as SUM and AVERAGE is computed.[9] However, NULL values are not considered in the aggregation, as we discussed in Section 7.1.7. It is worth emphasizing that the result of applying an aggregate function is a relation, not a scalar number—even if it has a single value. This makes the relational algebra a closed mathematical system.

---

[8]Note that this is an arbitrary notation, consistent with what SQL would do.

[9]In SQL, the option of eliminating duplicates before applying the aggregate function is available by including the keyword DISTINCT (see Section Section 4.4.4).

8.4.3 Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure.** This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor. This relationship is described by the foreign key Super_ssn of the EMPLOYEE relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of an employee e at all levels—that is, all employees e' directly supervised by e, all employees e'ℑ directly supervised by each employee e', all employees e''' directly supervised by each employee e'', and so on.

It is relatively straightforward in the relational algebra to specify all employees supervised by e *at a specific level* by joining the table with itself one or more times. However, it is difficult to specify all supervisees at *all* levels. For example, to specify the Ssns of all employees e' directly supervised—*at level one*—by the employee e whose name is 'James Borg' (see Figure 5.6), we can apply the following operation:

BORG_SSN $\leftarrow \pi_{Ssn}(\sigma_{Fname='James'\ \textbf{AND}\ Lname='Borg'}(EMPLOYEE))$
SUPERVISION(Ssn1, Ssn2) $\leftarrow \pi_{Ssn,Super\_ssn}(EMPLOYEE)$
RESULT1(Ssn) $\leftarrow \pi_{Ssn1}(SUPERVISION \bowtie_{Ssn2=Ssn}BORG\_SSN)$

To retrieve all employees supervised by Borg at level 2—that is, all employees e'' supervised by some employee e' who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

RESULT2(Ssn) $\leftarrow \pi_{Ssn1}(SUPERVISION \bowtie_{Ssn2=Ssn}RESULT1)$

To get both sets of employees supervised at levels 1 and 2 by 'James Borg', we can apply the UNION operation to the two results, as follows:

RESULT $\leftarrow$ RESULT2 $\cup$ RESULT1

The results of these queries are illustrated in Figure 8.11. Although it is possible to retrieve employees at each level and then take their UNION, we cannot, in general, specify a query such as "retrieve the supervisees of 'James Borg' at all levels" without utilizing a looping mechanism unless we know the maximum number of levels.[10] An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

8.4.4 OUTER JOIN Operations

Next, we discuss some additional extensions to the JOIN operation that are necessary to specify certain types of queries. The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN

---

[10]The SQL3 standard includes syntax for recursive closure.

SUPERVISION

(Borg's Ssn is **888665555**)

(Ssn)   (Super_ssn)

Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by
Borg's subordinates)

RESULT

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 ∪ RESULT2)

Figure 8.11
A two-level recursive query.

operation $R * S$, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result. Tuples with NULL values in the join attributes are also eliminated. This type of join, where tuples with no match are eliminated, is known as an **inner join**. The join operations we described earlier in Section 8.3 are all inner joins. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values. For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*; if they do not manage one, we can indicate it

Figure 8.12

The result of a LEFT OUTER JOIN operation.

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by ⋈, to retrieve the result as follows:

TEMP ← (EMPLOYEE ⋈ $_{Ssn=Mgr\_ssn}$DEPARTMENT)
RESULT ← $\pi_{Fname, Minit, Lname, Dname}$(TEMP)

The LEFT OUTER JOIN operation keeps every tuple in the *first*, or *left*, relation R in $R ⋈ S$; if no matching tuple is found in S, then the attributes of S in the join result are filled or *padded* with NULL values. The result of these operations is shown in Figure 8.12.

A similar operation, **RIGHT OUTER JOIN**, denoted by ⋈, keeps every tuple in the *second*, or right, relation S in the result of $R ⋈ S$. A third operation, **FULL OUTER JOIN**, denoted by ⋈, keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed. The three outer join operations are part of the SQL2 standard (see Section 7.1.6). These operations were provided later as an extension of relational algebra in response to the typical need in business applications to show related information from multiple tables exhaustively. Sometimes a complete reporting of data from multiple tables is required whether or not there are matching values.

8.4.5 The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations that have some common attributes, but are *not union (type) compatible*. This operation will take the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible**, meaning that only some of their attributes, say X, are union compatible. The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation $T(X, Y, Z)$. It is therefore the same as a FULL OUTER JOIN on the common attributes.

Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X] = t_2[X]$. These will be combined (unioned) into a single tuple in t. Tuples in either relation that have no matching tuple in the other relation are padded with NULL values. For example, an

OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, Ssn, Department, Advisor) and INSTRUCTOR(Name, Ssn, Department, Rank). Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department. The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

> STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)

All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute. A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes.[11]

Notice that the same person may still appear twice in the result. For example, we could have a graduate student in the Mathematics department who is an instructor in the Computer Science department. Although the two tuples representing that person in STUDENT and INSTRUCTOR will have the same (Name, Ssn) values, they will not agree on the Department value, and so will not be matched. This is because Department has two different meanings in STUDENT (the department where the person studies) and INSTRUCTOR (the department where the person is employed as an instructor). If we wanted to apply the OUTER UNION based on the same (Name, Ssn) combination only, we should rename the Department attribute in each table to reflect that they have different meanings and designate them as not being part of the union-compatible attributes. For example, we could rename the attributes as MajorDept in STUDENT and WorkDept in INSTRUCTOR.

8.5 Examples of Queries in Relational Algebra

The following are additional examples to illustrate the use of the relational algebra operations. All examples refer to the database in Figure 5.6. In general, the same query can be stated in numerous ways using the various operations. We will state each query in one way and leave it to the reader to come up with equivalent formulations.

> **Query 1.** Retrieve the name and address of all employees who work for the 'Research' department.

RESEARCH_DEPT ← $\sigma_{Dname='Research'}$(DEPARTMENT)
RESEARCH_EMPS ← (RESEARCH_DEPT ⋈ $_{Dnumber=Dno}$EMPLOYEE)
RESULT ← $\pi_{Fname, Lname, Address}$(RESEARCH_EMPS)

As a single in-line expression, this query becomes:

$\pi_{Fname, Lname, Address}$ ($\sigma_{Dname='Research'}$(DEPARTMENT ⋈ $_{Dnumber=Dno}$(EMPLOYEE)))

---

[11]Note that OUTER UNION is equivalent to a FULL OUTER JOIN if the join attributes are *all* the common attributes of the two relations.

This query could be specified in other ways; for example, the order of the JOIN and SELECT operations could be reversed, or the JOIN could be replaced by a NATURAL JOIN after renaming one of the join attributes to match the other join attribute name.

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

STAFFORD_PROJS ← $\sigma_{Plocation='Stafford'}$(PROJECT)
CONTR_DEPTS ← (STAFFORD_PROJS ⋈ $_{Dnum=Dnumber}$DEPARTMENT)
PROJ_DEPT_MGRS ← (CONTR_DEPTS ⋈ $_{Mgr\_ssn=Ssn}$EMPLOYEE)
RESULT ← $\pi_{Pnumber, Dnum, Lname, Address, Bdate}$(PROJ_DEPT_MGRS)

In this example, we first select the projects located in Stafford, then join them with their controlling departments, and then join the result with the department managers. Finally, we apply a project operation on the desired attributes.

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

DEPT5_PROJS ← $\rho_{(Pno)}(\pi_{Pnumber}(\sigma_{Dnum=5}$(PROJECT)))
EMP_PROJ ← $\rho_{(Ssn, Pno)}(\pi_{Essn, Pno}$(WORKS_ON))
RESULT_EMP_SSNS ← EMP_PROJ ÷ DEPT5_PROJS
RESULT ← $\pi_{Lname, Fname}$(RESULT_EMP_SSNS * EMPLOYEE)

In this query, we first create a table DEPT5_PROJS that contains the project numbers of all projects controlled by department 5. Then we create a table EMP_PROJ that holds (Ssn, Pno) tuples, and apply the division operation. Notice that we renamed the attributes so that they will be correctly used in the division operation. Finally, we join the result of the division, which holds only Ssn values, with the EMPLOYEE table to retrieve the Fname, Lname attributes from EMPLOYEE.

Query 4. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

SMITHS(Essn) ← π_{Ssn} ($\sigma_{Lname='Smith'}$(EMPLOYEE))
SMITH_WORKER_PROJS ← π_{Pno}(WORKS_ON * SMITHS)
MGRS ← $\pi_{Lname, Dnumber}$(EMPLOYEE ⋈ $_{Ssn=Mgr\_ssn}$DEPARTMENT)
SMITH_MANAGED_DEPTS(Dnum) ← $\pi_{Dnumber}$ ($\sigma_{Lname='Smith'}$(MGRS))
SMITH_MGR_PROJS(Pno) ← $\pi_{Pnumber}$(SMITH_MANAGED_DEPTS * PROJECT)
RESULT ← (SMITH_WORKER_PROJS ∪ SMITH_MGR_PROJS)

In this query, we retrieved the project numbers for projects that involve an employee named Smith as a worker in SMITH_WORKER_PROJS. Then we retrieved the project numbers for projects that involve an employee named Smith as manager of the department that controls the project in SMITH_MGR_PROJS. Finally, we applied the

UNION operation on SMITH_WORKER_PROJS and SMITH_MGR_PROJS. As a single in-line expression, this query becomes:

π_{Pno} (WORKS_ON $\bowtie_{Essn=Ssn}$ (π_{Ssn} ($\sigma_{Lname='Smith'}$(EMPLOYEE)))) $\cup$ π_{Pno}
(($\pi_{Dnumber}$ ($\sigma_{Lname='Smith'}$($\pi_{Lname, Dnumber}$(EMPLOYEE)))) $\bowtie$
$_{Ssn=Mgr\_ssn}$DEPARTMENT)) $\bowtie_{Dnum-ber=Dnum}$PROJECT)

Query 5. List the names of all employees with two or more dependents.

Strictly speaking, this query cannot be done in the *basic* (*original*) *relational algebra*. We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* Dependent_name values.

$T1$(Ssn, No_of_dependents)$\leftarrow$ $_{Essn}$ $\Im$ $_{COUNT\ Dependent\_name}$(DEPENDENT)
$T2 \leftarrow \sigma_{No\_of\_dependents>2}(T1)$
RESULT $\leftarrow \pi_{Lname,\ Fname}(T2 * $ EMPLOYEE)

Query 6. Retrieve the names of employees who have no dependents.

This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

ALL_EMPS $\leftarrow \pi_{Ssn}$(EMPLOYEE)
EMPS_WITH_DEPS(Ssn) $\leftarrow \pi_{Essn}$(DEPENDENT)
EMPS_WITHOUT_DEPS $\leftarrow$ (ALL_EMPS – EMPS_WITH_DEPS)
RESULT $\leftarrow \pi_{Lname,\ Fname}$(EMPS_WITHOUT_DEPS * EMPLOYEE)

We first retrieve a relation with all employee Ssns in ALL_EMPS. Then we create a table with the Ssns of employees who have at least one dependent in EMPS_WITH_DEPS. Then we apply the SET DIFFERENCE operation to retrieve employees Ssns with no dependents in EMPS_WITHOUT_DEPS, and finally join this with EMPLOYEE to retrieve the desired attributes. As a single in-line expression, this query becomes:

$\pi_{Lname,\ Fname}$((π_{Ssn}(EMPLOYEE) – ρ_{Ssn}(π_{Essn}(DEPENDENT))) * EMPLOYEE)

Query 7. List the names of managers who have at least one dependent.

MGRS(Ssn) $\leftarrow \pi_{Mgr\_ssn}$(DEPARTMENT)
EMPS_WITH_DEPS(Ssn) $\leftarrow \pi_{Essn}$(DEPENDENT)
MGRS_WITH_DEPS $\leftarrow$ (MGRS $\cap$ EMPS_WITH_DEPS)
RESULT $\leftarrow \pi_{Lname,\ Fname}$(MGRS_WITH_DEPS * EMPLOYEE)

In this query, we retrieve the Ssns of managers in MGRS, and the Ssns of employees with at least one dependent in EMPS_WITH_DEPS, then we apply the SET INTERSECTION operation to get the Ssns of managers who have at least one dependent.

As we mentioned earlier, the same query can be specified in many different ways in relational algebra. In particular, the operations can often be applied in various orders. In addition, some operations can be used to replace others; for example, the

INTERSECTION operation in Q7 can be replaced by a NATURAL JOIN. As an exercise, try to do each of these sample queries using different operations.[12] We showed how to write queries as single relational algebra expressions for queries Q1, Q4, and Q6. Try to write the remaining queries as single expressions. In Chapters 6 and 7 and in Sections 8.6 and 8.7, we show how these queries are written in other relational languages.

8.6 The Tuple Relational Calculus

In this and the next section, we introduce another formal query language for the relational model called **relational calculus**. This section introduces the language known as **tuple relational calculus**, and Section 8.7 introduces a variation called **domain relational calculus**. In both variations of relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how, or *in what order*, to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request *in a particular order* of applying the operations; thus, it can be considered as a **procedural** way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query. A calculus expression may be written in different ways, but the way it is written has no bearing on how a query should be evaluated.

It has been shown that any retrieval that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the languages is *identical*. This led to the definition of the concept of a *relationally complete* language. A relational query language *L* is considered **relationally complete** if we can express in *L* any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive power of high-level query languages. However, as we saw in Section 8.4, certain frequently required queries in database applications cannot be expressed in basic relational algebra or calculus. Most relational query languages are relationally complete but have *more expressive power* than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering. As we mentioned in the introduction to this chapter, the relational calculus is important for two reasons. First, it has a firm basis in mathematical logic. Second, the standard query language (SQL) for RDBMSs has its basic foundation in the tuple relational calculus.

Our examples refer to the database shown in Figures 5.6 and 5.7. We will use the same queries that were used in Section 8.5. Sections 8.6.6, 8.6.7, and 8.6.8 discuss dealing with universal quantifiers and safety of expression issues. Students interested in a basic introduction to tuple relational calculus may skip these sections.

---

[12]When queries are optimized (see Chapters 18 and 19), the system will choose a particular sequence of operations that corresponds to an execution strategy that can be executed efficiently.

8.6.1 Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form:

$\{t \mid \text{COND}(t)\}$

where t is a tuple variable and COND(t) is a conditional (Boolean) expression involving t that evaluates to either TRUE or FALSE for different assignments of tuples to the variable t. The result of such a query is the set of all tuples t that evaluate COND(t) to TRUE. These tuples are said to **satisfy** COND(t). For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

$\{t \mid \text{EMPLOYEE}(t)$ **AND** $t.\text{Salary}{>}50000\}$

The condition EMPLOYEE(t) specifies that the **range relation** of tuple variable t is EMPLOYEE. Each EMPLOYEE tuple t that satisfies the condition t.Salary>50000 will be retrieved. Notice that t.Salary references attribute Salary of tuple variable t; this notation resembles how attribute names are qualified with relation names or aliases in SQL, as we saw in Chapter 6. In the notation of Chapter 5, t.Salary is the same as writing t[Salary].

The previous query retrieves all attribute values for each selected EMPLOYEE tuple t. To retrieve only *some* of the attributes—say, the first and last names—we write

$t.\text{Fname}, t.\text{Lname} \mid \text{EMPLOYEE}(t)$ **AND** $t.\text{Salary}{>}50000\}$

Informally, we need to specify the following information in a tuple relational calculus expression:

- For each tuple variable t, the **range relation** R of t. This value is specified by a condition of the form $R(t)$. If we do not specify a range relation, then the variable t will range over all possible tuples "in the universe" as it is not restricted to any one relation.

- A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.

- A set of attributes to be retrieved, the **requested attributes**. The values of these attributes are retrieved for each selected combination of tuples.

Before we discuss the formal syntax of tuple relational calculus, consider another query.

Query 0. Retrieve the birth date and address of the employee (or employees) whose name is John B. Smith.

Q0: $\{t.\text{Bdate}, t.\text{Address} \mid \text{EMPLOYEE}(t)$ **AND** $t.\text{Fname}=\text{'John'}$ **AND** $t.\text{Minit}=\text{'B'}$
   AND $t.\text{Lname}=\text{'Smith'}\}$

In tuple relational calculus, we first specify the requested attributes t.Bdate and t.Address for each selected tuple t. Then we specify the condition for selecting a tuple following the bar (|)—namely, that t be a tuple of the EMPLOYEE relation whose Fname, Minit, and Lname attribute values are 'John', 'B', and 'Smith', respectively.

8.6.2 Expressions and Formulas in Tuple Relational Calculus

A general **expression** of the tuple relational calculus is of the form

$$\{t_1.A_j, t_2.A_k, \dots , t_n.A_m \mid \textbf{COND}(t_1, t_2, ..., t_n, t_{n+1}, t_{n+2}, ..., t_{n+m})\}$$

where $t_1, t_2, \dots , t_n, t_{n+1}, \dots , t_{n+}m$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula**[13] of the tuple relational calculus. A formula is made up of predicate calculus **atoms**, which can be one of the following:

1. An atom of the form $R(t_i)$, where R is a relation name and t_i is a tuple variable. This atom identifies the range of the tuple variable t_i as the relation whose name is R. It evaluates to TRUE if t_i is a tuple in the relation R, and evaluates to FALSE otherwise.

2. An atom of the form $t_i.A$ **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \le, >, \ge, \ne\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, and B is an attribute of the relation on which t_j ranges.

3. An atom of the form $t_i.A$ **op** c or c **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \le, >, \ge, \ne\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, B is an attribute of the relation on which t_j ranges, and c is a constant value.

Each of the preceding atoms evaluates to either TRUE or FALSE for a specific combination of tuples; this is called the **truth value** of an atom. In general, a tuple variable t ranges over all possible tuples *in the universe*. For atoms of the form $R(t)$, if t is assigned to a tuple that is a *member of the specified relation R*, the atom is TRUE; otherwise, it is FALSE. In atoms of types 2 and 3, if the tuple variables are assigned to tuples such that the values of the specified attributes of the tuples satisfy the condition, then the atom is TRUE.

A **formula** (Boolean condition) is made up of one or more atoms connected via the logical operators **AND**, **OR**, and **NOT** and is defined recursively by Rules 1 and 2 as follows:

- *Rule 1*: Every atom is a formula.
- *Rule 2*: If F_1 and F_2 are formulas, then so are $(F_1$ **AND** $F_2)$, $(F_1$ **OR** $F_2)$, **NOT** (F_1), and **NOT** (F_2). The truth values of these formulas are derived from their component formulas F_1 and F_2 as follows:

---

[13]Also called a **well-formed formula**, or **WFF**, in mathematical logic.

    a. (F_1 **AND** F_2) is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.

    b. (F_1 **OR** F_2) is FALSE if both F_1 and F_2 are FALSE; otherwise, it is TRUE.

    c. **NOT** (F_1) is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.

    d. **NOT** (F_2) is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

8.6.3 The Existential and Universal Quantifiers

In addition, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** ($\forall$) and the **existential quantifier** ($\exists$). Truth values for formulas with quantifiers are described in Rules 3 and 4 below; first, however, we need to define the concepts of free and bound tuple variables in a formula. Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an ($\exists t$) or ($\forall t$) clause; otherwise, it is free. Formally, we define a tuple variable in a formula as **free** or **bound** according to the following rules:

- An occurrence of a tuple variable in a formula F that *is an atom* is free in F.
- An occurrence of a tuple variable t is free or bound in a formula made up of logical connectives—(F_1 **AND** F_2), (F_1 **OR** F_2), **NOT**(F_1), and **NOT**(F_2)—depending on whether it is free or bound in F_1 or F_2 (if it occurs in either). Notice that in a formula of the form $F = (F_1$ **AND** $F_2)$ or $F = (F_1$ **OR** $F_2)$, a tuple variable may be free in F_1 and bound in F_2, or vice versa; in this case, one occurrence of the tuple variable is bound and the other is free in F.
- All *free* occurrences of a tuple variable t in F are **bound** in a formula F' of the form $F' = (\exists t)(F)$ or $F' = (\forall t)(F)$. The tuple variable is bound to the quantifier specified in F'. For example, consider the following formulas:

  F_1: d.Dname = 'Research'
  F_2: ($\exists t$)(d.Dnumber = t.Dno)
  F_3: ($\forall d$)(d.Mgr_ssn = '333445555')

The tuple variable d is free in both F_1 and F_2, whereas it is bound to the ($\forall$) quantifier in F_3. Variable t is bound to the ($\exists$) quantifier in F_2.

We can now give Rules 3 and 4 for the definition of a formula we started earlier:

- *Rule 3*: If F is a formula, then so is ($\exists t$)(F), where t is a tuple variable. The formula ($\exists t$)(F) is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurrences of t in F; otherwise, ($\exists t$)(F) is FALSE.
- *Rule 4*: If F is a formula, then so is ($\forall t$)(F), where t is a tuple variable. The formula ($\forall t$)(F) is TRUE if the formula F evaluates to TRUE for *every tuple* (in the universe) assigned to free occurrences of t in F; otherwise, ($\forall t$)(F) is FALSE.

The ($\exists$) quantifier is called an existential quantifier because a formula ($\exists t$)(F) is TRUE if *there exists* some tuple that makes F TRUE. For the universal quantifier, ($\forall t$)(F) is TRUE if every possible tuple that can be assigned to free occurrences of t in F is substituted for t, and F is TRUE for *every such substitution*. It is called the universal or *for all* quantifier because every tuple in *the universe of* tuples must make F TRUE to make the quantified formula TRUE.

8.6.4 Sample Queries in Tuple Relational Calculus

We will use some of the same queries from Section 8.5 to give a flavor of how the same queries are specified in relational algebra and in relational calculus. Notice that some queries are easier to specify in the relational algebra than in the relational calculus, and vice versa.

Query 1. List the name and address of all employees who work for the 'Research' department.

Q1: $\{t.\text{Fname}, t.\text{Lname}, t.\text{Address} \mid \text{EMPLOYEE}(t) \textbf{ AND } (\exists d)(\text{DEPARTMENT}(d) \textbf{ AND } d.\text{Dname}=\text{'Research'} \textbf{ AND } d.\text{Dnumber}=t.\text{Dno})\}$

The *only free tuple variables* in a tuple relational calculus expression should be those that appear to the left of the bar (|). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified after the bar in Q1, the attributes Fname, Lname, and Address are retrieved for each such tuple. The conditions EMPLOYEE(t) and DEPARTMENT(d) specify the range relations for t and d. The condition d.Dname = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra, whereas the condition d.Dnumber = t.Dno is a **join condition** and is similar in purpose to the (INNER) JOIN operation (see Section 8.3).

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

Q2: $\{p.\text{Pnumber}, p.\text{Dnum}, m.\text{Lname}, m.\text{Bdate}, m.\text{Address} \mid \text{PROJECT}(p) \textbf{ AND } \text{EMPLOYEE}(m) \textbf{ AND } p.\text{Plocation}=\text{'Stafford'} \textbf{ AND } ((\exists d)(\text{DEPARTMENT}(d) \textbf{ AND } p.\text{Dnum}=d.\text{Dnumber} \textbf{ AND } d.\text{Mgr\_ssn}=m.\text{Ssn}))\}$

In Q2 there are two free tuple variables, p and m. Tuple variable d is bound to the existential quantifier. The query condition is evaluated for every combination of tuples assigned to p and m, and out of all possible combinations of tuples to which p and m are bound, only the combinations that satisfy the condition are selected.

Several tuple variables in a query can range over the same relation. For example, to specify Q8—for each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor—we specify two tuple variables e and s that both range over the EMPLOYEE relation:

Q8: $\{e.\text{Fname}, e.\text{Lname}, s.\text{Fname}, s.\text{Lname} \mid \text{EMPLOYEE}(e) \textbf{ AND } \text{EMPLOYEE}(s) \textbf{ AND } e.\text{Super\_ssn}=s.\text{Ssn}\}$

Query 3′. List the name of each employee who works on *some* project controlled by department number 5. This is a variation of Q3 in which *all* is changed to *some*. In this case we need two join conditions and two existential quantifiers.

Q0′: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e) \textbf{ AND } ((\exists x)(\exists w)(\text{PROJECT}(x) \textbf{ AND } \text{WORKS\_ON}(w) \textbf{ AND } x.\text{Dnum}=5 \textbf{ AND } w.\text{Essn}=e.\text{Ssn} \textbf{ AND } x.\text{Pnumber}=w.\text{Pno}))\}$

Query 4. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as manager of the controlling department for the project.

Q4: { p.Pnumber | PROJECT(p) **AND** ((($\exists e$)($\exists w$)(EMPLOYEE(e)
   AND WORKS_ON(w) **AND** w.Pno=p.Pnumber
   AND e.Lname='Smith' **AND** e.Ssn=w.Essn))
   OR
   (($\exists m$)($\exists d$)(EMPLOYEE(m) **AND** DEPARTMENT(d)
   AND p.Dnum=d.Dnumber **AND** d.Mgr_ssn=m.Ssn
   AND m.Lname='Smith')))}

Compare this with the relational algebra version of this query in Section 8.5. The UNION operation in relational algebra can usually be substituted with an OR connective in relational calculus.

8.6.5 Notation for Query Graphs

In this section, we describe a notation that has been proposed to represent relational calculus queries that do not involve complex quantification in a graphical form. These types of queries are known as **select-project-join queries** because they only involve these three relational algebra operations. The notation may be expanded to more general queries, but we do not discuss these extensions here. This graphical representation of a query is called a **query graph**. Figure 8.13 shows the query graph for Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges** (the lines that connect the nodes), as shown in Figure 8.13. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

The query graph representation does not indicate a particular order to specify which operations to perform first, and is hence a more neutral representation of a select-project-join query than the query tree representation (see Section 8.3.5), where the order of execution is implicitly specified. There is only a single query graph corresponding to each query. Although some query optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because,

[P.Pnumber,P.Dnum]   [E.Lname,E.address,E.Bdate]

Figure 8.13
Query graph for Q2.

in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

In the next section we discuss the relationship between the universal and existential quantifiers and show how one can be transformed into the other.

8.6.6 Transforming the Universal and Existential Quantifiers

We now introduce some well-known transformations from mathematical logic that relate the universal and existential quantifiers. It is possible to transform a universal quantifier into an existential quantifier, and vice versa, to get an equivalent expression. One general transformation can be described informally as follows: Transform one type of quantifier into the other with negation (preceded by **NOT**); **AND** and **OR** replace one another; a negated formula becomes unnegated; and an unnegated formula becomes negated. Some special cases of this transformation can be stated as follows, where the $\equiv$ symbol stands for **equivalent to**:

$(\forall x) (P(x)) \equiv$ **NOT** $(\exists x) ($**NOT** $(P(x)))$
$(\exists x) (P(x)) \equiv$ **NOT** $(\forall x) ($**NOT** $(P(x)))$
$(\forall x) (P(x)$ **AND** $Q(x)) \equiv$ **NOT** $(\exists x) ($**NOT** $(P(x))$ **OR NOT** $(Q(x)))$
$(\forall x) (P(x)$ **OR** $Q(x)) \equiv$ **NOT** $(\exists x) ($**NOT** $(P(x))$ **AND NOT** $(Q(x)))$
$(\exists x) (P(x))$ **OR** $Q(x)) \equiv$ **NOT** $(\forall x) ($**NOT** $(P(x))$ **AND NOT** $(Q(x)))$
$(\exists x) (P(x)$ **AND** $Q(x)) \equiv$ **NOT** $(\forall x) ($**NOT** $(P(x))$ **OR NOT** $(Q(x)))$

Notice also that the following is TRUE, where the $\Rightarrow$ symbol stands for **implies**:

$(\forall x)(P(x)) \Rightarrow (\exists x)(P(x))$
NOT $(\exists x)(P(x)) \Rightarrow$ **NOT** $(\forall x)(P(x))$

8.6.7 Using the Universal Quantifier in Queries

Whenever we use a universal quantifier, it is quite judicious to follow a few rules to ensure that our expression makes sense. We discuss these rules with respect to the query Q3.

Query 3. List the names of employees who work on *all* the projects controlled by department number 5. One way to specify this query is to use the universal quantifier as shown:

Q3: $\{e.$Lname, $e.$Fname $|$ EMPLOYEE(e) **AND** $((\forall x)($**NOT**$($PROJECT$(x))$ **OR NOT** $(x.$Dnum=5$)$ **OR** $((\exists w)($WORKS_ON(w) **AND** $w.$Essn=$e.$Ssn **AND** $x.$Pnumber=$w.$Pno$))))\}$

We can break up Q3 into its basic components as follows:

Q3: $\{e.$Lname, $e.$Fname $|$ EMPLOYEE(e) **AND** $F'\}$
$F' = ((\forall x)($**NOT**$($PROJECT$(x))$ **OR** $F_1))$
$F_1 =$ **NOT**$(x.$Dnum=5$)$ **OR** F_2
$F_2 = ((\exists w)($WORKS_ON(w) **AND** $w.$Essn=$e.$Ssn **AND** $x.$Pnumber=$w.$Pno$))$

We want to make sure that a selected employee *e* works on *all the projects* controlled by department 5, but the *definition of universal quantifier* says that to make the quantified formula TRUE, *the inner formula* must be TRUE *for all tuples in the universe*. The trick is to exclude from the universal quantification all tuples that we are not interested in by making the condition TRUE *for all such tuples*. This is necessary because a universally quantified tuple variable, such as *x* in Q3, must evaluate to TRUE *for every possible tuple* assigned to it to make the quantified formula TRUE.

The first tuples to exclude (by making them evaluate automatically to TRUE) are those that are not in the relation *R* of interest. In Q3, using the expression **NOT**(PROJECT(*x*)) inside the universally quantified formula evaluates to TRUE all tuples *x* that are not in the PROJECT relation. Then we exclude the tuples we are not interested in from *R* itself. In Q3, using the expression **NOT**(*x*.Dnum=5) evaluates to TRUE all tuples *x* that are in the PROJECT relation but are not controlled by department 5. Finally, we specify a condition F_2 that must hold on all the remaining tuples in *R*. Hence, we can explain Q3 as follows:

1. For the formula $F' = (\forall x)(F)$ to be TRUE, we must have the formula *F* be TRUE *for all tuples in the universe that can be assigned to x*. However, in Q3 we are only interested in *F* being TRUE for all tuples of the PROJECT relation that are controlled by department 5. Hence, the formula *F* is of the form (**NOT**(PROJECT(*x*)) **OR** F_1). The '**NOT** (*PROJECT*(*x*)) **OR** …' condition is TRUE for all tuples *not in the* PROJECT *relation* and has the effect of eliminating these tuples from consideration in the truth value of F_1. For every tuple in the PROJECT relation, F_1 must be TRUE if F' is to be TRUE.

2. Using the same line of reasoning, we do not want to consider tuples in the PROJECT relation that are not controlled by department number 5, since we are only interested in PROJECT tuples whose Dnum=5. Therefore, we can write:

   IF (*x*.Dnum=5) **THEN** F_2

   which is equivalent to

   (**NOT** (*x*.Dnum=5) **OR** F_2)

3. Formula F_1, hence, is of the form **NOT**(*x*.Dnum=5) **OR** F_2. In the context of Q3, this means that, for a tuple *x* in the PROJECT relation, either its Dnum≠5 or it must satisfy F_2.

4. Finally, F_2 gives the condition that we want to hold for a selected EMPLOYEE tuple: that the employee works on *every* PROJECT *tuple that has not been excluded yet*. Such employee tuples are selected by the query.

In English, Q3 gives the following condition for selecting an EMPLOYEE tuple *e*: For every tuple *x* in the PROJECT relation with *x*.Dnum=5, there must exist a tuple *w* in WORKS_ON such that *w*.Essn=*e*.Ssn and *w*.Pno=*x*.Pnumber. This is equivalent to saying that EMPLOYEE *e* works on every PROJECT *x* in DEPARTMENT number 5. (Whew!)

Using the general transformation from universal to existential quantifiers given in Section 8.6.6, we can rephrase the query in Q3 as shown in Q3A, which uses a negated existential quantifier instead of the universal quantifier:

Q3A: {e.Lname, e.Fname | EMPLOYEE(e) **AND** (**NOT** ($\exists x$) (PROJECT(x) **AND**
(x.Dnum=5) and (**NOT** ($\exists w$)(WORKS_ON(w) **AND** w.Essn=e.Ssn
AND x.Pnumber=w.Pno)))))}

We now give some additional examples of queries that use quantifiers.

Query 6. List the names of employees who have no dependents.

Q6: {e.Fname, e.Lname | EMPLOYEE(e) **AND** (**NOT** ($\exists d$)(DEPENDENT(d)
AND e.Ssn=d.Essn))}

Using the general transformation rule, we can rephrase Q6 as follows:

Q6A: {e.Fname, e.Lname | EMPLOYEE(e) **AND** (($\forall d$)(**NOT**(DEPENDENT(d))
OR NOT(e.Ssn=d.Essn)))}

Query 7. List the names of managers who have at least one dependent.

Q7: {e.Fname, e.Lname | EMPLOYEE(e) **AND** (($\exists d$)($\exists \rho$)(DEPARTMENT(d)
AND DEPENDENT(ρ) **AND** e.Ssn=d.Mgr_ssn **AND** ρ.Essn=e.Ssn))}

This query is handled by interpreting *managers who have at least one dependent* as *managers for whom there exists some dependent.*

8.6.8 Safe Expressions

Whenever we use universal quantifiers, existential quantifiers, or negation of predi-cates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe**. For example, the expression

{t | **NOT** (EMPLOYEE(t))}

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. If we follow the rules for Q3 discussed earlier, we will get a safe expression when using universal quantifiers. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression:* This is the set of all values that either appear as constant values in the expression or exist in any tuple in the relations referenced in the expression. For example, the domain of {t | **NOT**(EMPLOYEE(t))} is the set of all attribute values appearing in some tuple of the EMPLOYEE relation (for any attribute). The domain of the expression Q3A would include all values appearing in EMPLOYEE, PROJECT, and WORKS_ON (unioned with the value 5 appearing in the query itself).

An expression is said to be **safe** if all values in its result are from the domain of the expression. Notice that the result of {t | **NOT**(EMPLOYEE(t))} is unsafe, since it will,

in general, include tuples (and hence values) from outside the EMPLOYEE relation; such values are not in the domain of the expression. All of our other examples are safe expressions.

8.7 The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply **domain calculus**. Historically, while SQL (see Chapters 6 and 7), which was based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language called QBE (Query-By-Example), which is related to domain calculus, was being developed almost concurrently at the IBM T. J. Watson Research Center in Yorktown Heights, New York. The formal specification of the domain calculus was proposed after the development of the QBE language and system.

Domain calculus differs from tuple calculus in the *type of variables* used in formulas: Rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute. An expression of the domain calculus is of the form

$$\{x_1, x_2, ..., x_n \mid \mathsf{COND}(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m})\}$$

where $x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}$ are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

A formula is made up of **atoms**. The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form $R(x_1, x_2, \dots, x_j)$, where R is the name of a relation of degree j and each x_i, $1 \le i \le j$, is a domain variable. This atom states that a list of values of $<x_1, x_2, \dots, x_j>$ must be a tuple in the relation whose name is R, where x_i is the value of the ith attribute value of the tuple. To make a domain calculus expression more concise, we can *drop the commas* in a list of variables; thus, we can write:

    $$\{x_1, x_2, ..., x_n \mid R(x_1\ x_2\ x_3)\ \mathsf{AND}\ ...\}$$

    instead of:

    $$\{x_1, x_2, \dots, x_n \mid R(x_1, x_2, x_3)\ \mathsf{AND}\ ...\}$$

2. An atom of the form x_i **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \le, >, \ge, \ne\}$, and x_i and x_j are domain variables.

3. An atom of the form x_i **op** c or c **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \le, >, \ge, \ne\}$, x_i and x_j are domain variables, and c is a constant value.

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the **truth values** of the atoms. In case 1, if the domain variables are

assigned values corresponding to a tuple of the specified relation R, then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here. Some examples of queries specified in the domain calculus follow. We will use low-ercase letters l, m, n, … , x, y, z for domain variables.

> **Query 0.** List the birth date and address of the employee whose name is 'John B. Smith'.

> **Q0:** $\{u, v \mid (\exists q) (\exists r) (\exists s) (\exists t) (\exists w) (\exists x) (\exists y) (\exists z)$
> (EMPLOYEE($qrstuvwxyz$) **AND** q='John' **AND** r='B' **AND** s='Smith')$\}$

We need ten variables for the EMPLOYEE relation, one to range over each of the domains of attributes of EMPLOYEE in order. Of the ten variables q, r, s, … , z, only u and v are free, because they appear to the left of the bar and hence should not be bound to a quantifier. We first specify the *requested attributes*, Bdate and Address, by the free domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (|)—namely, that the sequence of values assigned to the variables $qrstuvwxyz$ be a tuple of the EMPLOYEE relation and that the values for q (Fname), r (Minit), and s (Lname) be equal to 'John', 'B', and 'Smith', respectively. For convenience, we will quantify only those variables *actually appearing in a condition* (these would be q, r, and s in Q0) in the rest of our examples.[14]

An alternative shorthand notation, used in QBE, for writing this query is to assign the constants 'John', 'B', and 'Smith' directly as shown in Q0A. Here, all variables not appearing to the left of the bar are implicitly existentially quantified:[15]

> **Q0A:** $\{u, v \mid$ EMPLOYEE('John', 'B', 'Smith', t, u, v, w, x, y, z)$\}$

> **Query 1.** Retrieve the name and address of all employees who work for the 'Research' department.

> **Q1:** $\{q, s, v \mid (\exists z) (\exists l) (\exists m)$ (EMPLOYEE($qrstuvwxyz$) **AND**
> DEPARTMENT($lmno$) **AND** l='Research' **AND** m=z)$\}$

A condition relating two domain variables that range over attributes from two rela-tions, such as $m = z$ in Q1, is a **join condition**, whereas a condition that relates a domain variable to a constant, such as l = 'Research', is a **selection condition**.

> **Query 2.** For every project located in 'Stafford', list the project number, the con-trolling department number, and the department manager's last name, birth date, and address.

---

[14]Quantifying only the domain variables actually used in conditions and specifying a predicate such as EMPLOYEE($qrstuvwxyz$) without separating domain variables with commas is an abbreviated notation used for convenience; it is not the correct formal notation.

[15]Again, this is not a formally accurate notation.

Q2: $\{i, k, s, u, v \mid (\exists j)(\exists m)(\exists n)(\exists t)(\text{PROJECT}(hijk)$ **AND**
$\quad$ EMPLOYEE(*qrstuvwxyz*) **AND** DEPARTMENT(*lmno*) **AND** $k=m$ **AND**
$\quad n=t$ **AND** $j=$'Stafford')$\}$

Query 6. List the names of employees who have no dependents.

Q6: $\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz)$ **AND**
$\quad$ (**NOT**$(\exists l)(\text{DEPENDENT}(lmnop)$ **AND** $t=l)))\}$

Q6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A: $\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz)$ **AND**
$\quad ((\forall l)(\textbf{NOT}(\text{DEPENDENT}(lmnop))$ **OR NOT**$(t=l))))\}$

Query 7. List the names of managers who have at least one dependent.

Q7: $\{s, q \mid (\exists t)(\exists j)(\exists l)(\text{EMPLOYEE}(qrstuvwxyz)$ **AND** DEPARTMENT(*hijk*)
$\quad$ **AND** DEPENDENT(*lmnop*) **AND** $t=j$ **AND** $l=t)\}$

As we mentioned earlier, it can be shown that any query that can be expressed in the basic relational algebra can also be expressed in the domain or tuple relational calculus. Also, any *safe expression* in the domain or tuple relational calculus can be expressed in the basic relational algebra.

The QBE language was based on the domain relational calculus, although this was realized later, after the domain calculus was formalized. QBE was one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the Query Management Facility (QMF) interface option to DB2. The basic ideas used in QBE have been applied in several other commercial products. Because of its important place in the history of relational languages, we have included an overview of QBE in Appendix C.

8.8 Summary

In this chapter we presented two formal languages for the relational model of data. They are used to manipulate relations and produce new relations as answers to queries. We discussed the relational algebra and its operations, which are used to specify a sequence of operations to specify a query. Then we introduced two types of relational calculi called tuple calculus and domain calculus.

In Sections 8.1 through 8.3, we introduced the basic relational algebra operations and illustrated the types of queries for which each is used. First, we discussed the unary relational operators SELECT and PROJECT, as well as the RENAME operation. Then, we discussed binary set theoretic operations requiring that relations on which they are applied be union (or type) compatible; these include UNION, INTERSECTION, and SET DIFFERENCE. The CARTESIAN PRODUCT operation is a set operation that can be used to combine tuples from two relations, producing all possible combinations. It is rarely used in practice; however, we showed how

CARTESIAN PRODUCT followed by SELECT can be used to define matching tuples from two relations and leads to the JOIN operation. Different JOIN operations called THETA JOIN, EQUIJOIN, and NATURAL JOIN were introduced. Query trees were introduced as a graphical representation of relational algebra queries, which can also be used as the basis for internal data structures that the DBMS can use to represent a query.

We discussed some important types of queries that *cannot* be stated with the basic relational algebra operations but are important for practical situations. We introduced GENERALIZED PROJECTION to use functions of attributes in the projection list and the AGGREGATE FUNCTION operation to deal with aggregate types of statistical requests that summarize the information in the tables. We discussed recursive queries, for which there is no direct support in the algebra but which can be handled in a step-by-step approach, as we demonstrated. Then we presented the OUTER JOIN and OUTER UNION operations, which extend JOIN and UNION and allow all information in source relations to be preserved in the result.

The last two sections described the basic concepts behind relational calculus, which is based on the branch of mathematical logic called predicate calculus. There are two types of relational calculi: (1) the tuple relational calculus, which uses tuple variables that range over tuples (rows) of relations, and (2) the domain relational calculus, which uses domain variables that range over domains (columns of relations). In relational calculus, a query is specified in a single declarative statement, without specifying any order or method for retrieving the query result. Hence, relational calculus is often considered to be a higher-level *declarative* language than the relational algebra, because a relational calculus expression states *what* we want to retrieve regardless of *how* the query may be executed.

We introduced query graphs as an internal representation for queries in relational calculus. We also discussed the existential quantifier ($\exists$) and the universal quantifier ($\forall$). We discussed the problem of specifying safe queries whose results are finite. We also discussed rules for transforming universal into existential quantifiers, and vice versa. It is the quantifiers that give expressive power to the relational calculus, making it equivalent to the basic relational algebra. There is no analog to grouping and aggregation functions in basic relational calculus, although some extensions have been suggested.

Review Questions

8.1. List the operations of relational algebra and the purpose of each.

8.2. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?

8.3. Discuss some types of queries for which renaming of attributes is necessary in order to specify the query unambiguously.

8.4. Discuss the various types of *inner join* operations. Why is theta join required?

8.5. What role does the concept of *foreign key* play when specifying the most common types of meaningful join operations?

8.6. What is the FUNCTION operation? For what is it used?

8.7. How are the OUTER JOIN operations different from the INNER JOIN operations? How is the OUTER UNION operation different from UNION?

8.8. In what sense does relational calculus differ from relational algebra, and in what sense are they similar?

8.9. How does tuple relational calculus differ from domain relational calculus?

8.10. Discuss the meanings of the existential quantifier ($\exists$) and the universal quantifier ($\forall$).

8.11. Define the following terms with respect to the tuple calculus: *tuple variable*, *range relation*, *atom*, *formula*, and *expression*.

8.12. Define the following terms with respect to the domain calculus: *domain variable*, *range relation*, *atom*, *formula*, and *expression*.

8.13. What is meant by a *safe expression* in relational calculus?

8.14. When is a query language called relationally complete?

Exercises

8.15. Show the result of each of the sample queries in Section 8.5 as it would apply to the database state in Figure 5.6.

8.16. Specify the following queries on the COMPANY relational database schema shown in Figure 5.5 using the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database state in Figure 5.6.

    a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.

    b. List the names of all employees who have a dependent with the same first name as themselves.

    c. Find the names of all employees who are directly supervised by 'Franklin Wong'.

    d. For each project, list the project name and the total hours per week (by all employees) spent on that project.

    e. Retrieve the names of all employees who work on every project.

    f. Retrieve the names of all employees who do not work on any project.

    g. For each department, retrieve the department name and the average salary of all employees working in that department.

    h. Retrieve the average salary of all female employees.

    i. Find the names and addresses of all employees who work on at least one project located in Houston but whose department has no location in Houston.

    j. List the last names of all department managers who have no dependents.

8.17. Consider the AIRLINE relational database schema shown in Figure 5.8, which was described in Exercise 5.12. Specify the following queries in relational algebra:

    a. For each flight, list the flight number, the departure airport for the first leg of the flight, and the arrival airport for the last leg of the flight.

    b. List the flight numbers and weekdays of all flights or flight legs that depart from Houston Intercontinental Airport (airport code 'iah') and arrive in Los Angeles International Airport (airport code 'lax').

    c. List the flight number, departure airport code, scheduled departure time, arrival airport code, scheduled arrival time, and weekdays of all flights or flight legs that depart from some airport in the city of Houston and arrive at some airport in the city of Los Angeles.

    d. List all fare information for flight number 'co197'.

    e. Retrieve the number of available seats for flight number 'co197' on '2009-10-09'.

8.18. Consider the LIBRARY relational database schema shown in Figure 8.14, which is used to keep track of books, borrowers, and book loans. Referential integrity constraints are shown as directed arcs in Figure 8.14, as in the notation of Figure 5.7. Write down relational expressions for the following queries:

    a. How many copies of the book titled *The Lost Tribe* are owned by the library branch whose name is 'Sharpstown'?

    b. How many copies of the book titled *The Lost Tribe* are owned by each library branch?

    c. Retrieve the names of all borrowers who do not have any books checked out.

    d. For each book that is loaned out from the Sharpstown branch and whose Due_date is today, retrieve the book title, the borrower's name, and the borrower's address.

    e. For each library branch, retrieve the branch name and the total number of books loaned out from that branch.

    f. Retrieve the names, addresses, and number of books checked out for all borrowers who have more than five books checked out.

    g. For each book authored (or coauthored) by Stephen King, retrieve the title and the number of copies owned by the library branch whose name is Central.

8.19. Specify the following queries in relational algebra on the database schema given in Exercise 5.14:

BOOK

Book_id	Title	Publisher_name

BOOK_AUTHORS

Book_id	Author_name

PUBLISHER

Name	Address	Phone

BOOK_COPIES

Book_id	Branch_id	No_of_copies

BOOK_LOANS

Book_id	Branch_id	Card_no	Date_out	Due_date

LIBRARY_BRANCH

Branch_id	Branch_name	Address

BORROWER

Card_no	Name	Address	Phone

Figure 8.14
A relational database schema for a LIBRARY database.

a. List the Order# and Ship_date for all orders shipped from Warehouse# W2.

b. List the WAREHOUSE information from which the CUSTOMER named Jose Lopez was supplied his orders. Produce a listing: Order#, Warehouse#.

c. Produce a listing Cname, No_of_orders, Avg_order_amt, where the middle column is the total number of orders by the customer and the last column is the average order amount for that customer.

d. List the orders that were not shipped within 30 days of ordering.

e. List the Order# for orders that were shipped from *all* warehouses that the company has in New York.

8.20. Specify the following queries in relational algebra on the database schema given in Exercise 5.15:

a. Give the details (all attributes of trip relation) for trips that exceeded $2,000 in expenses.

    b. Print the Ssns of salespeople who took trips to Honolulu.

    c. Print the total trip expenses incurred by the salesperson with SSN = '234-56-7890'.

8.21. Specify the following queries in relational algebra on the database schema given in Exercise 5.16:

    a. List the number of courses taken by all students named John Smith in Winter 2009 (i.e., Quarter=W09).

    b. Produce a list of textbooks (include Course#, Book_isbn, Book_title) for courses offered by the 'CS' department that have used more than two books.

    c. List any department that has all its adopted books published by 'Pearson Publishing'.

8.22. Consider the two tables $T1$ and $T2$ shown in Figure 8.15. Show the results of the following operations:

    a. $T1 \bowtie_{T1.P = T2.A} T2$

    b. $T1 \bowtie_{T1.Q = T2.B} T2$

    c. $T1 \bowtie_{T1.P = T2.A} T2$

    d. $T1 \bowtie_{T1.Q = T2.B} T2$

    e. $T1 \cup T2$

    f. $T1 \bowtie_{(T1.P = T2.A \text{ AND } T1.R = T2.C)} T2$

8.23. Specify the following queries in relational algebra on the database schema in Exercise 5.17:

    a. For the salesperson named 'Jane Doe', list the following information for all the cars she sold: Serial#, Manufacturer, Sale_price.

    b. List the Serial# and Model of cars that have no options.

    c. Consider the NATURAL JOIN operation between SALESPERSON and SALE. What is the meaning of a left outer join for these tables (do not change the order of relations)? Explain with an example.

    d. Write a query in relational algebra involving selection and one set operation and say in words what the query does.

8.24. Specify queries a, b, c, e, f, i, and j of Exercise 8.16 in both tuple and domain relational calculus.

8.25. Specify queries a, b, c, and d of Exercise 8.17 in both tuple and domain relational calculus.

Figure 8.15
A database state for the relations $T1$ and $T2$.

TABLE T1		
P	Q	R
10	a	5
15	b	8
25	a	6

TABLE T2		
A	B	C
10	b	6
25	c	3
10	b	5

8.26. Specify queries c, d, and f of Exercise 8.18 in both tuple and domain relational calculus.

8.27. In a tuple relational calculus query with n tuple variables, what would be the typical minimum number of join conditions? Why? What is the effect of having a smaller number of join conditions?

8.28. Rewrite the domain relational calculus queries that followed Q0 in Section 8.7 in the style of the abbreviated notation of Q0A, where the objective is to minimize the number of domain variables by writing constants in place of variables wherever possible.

8.29. Consider this query: Retrieve the Ssns of employees who work on at least those projects on which the employee with Ssn=123456789 works. This may be stated as (FORALL x) (IF P THEN Q), where

- x is a tuple variable that ranges over the PROJECT relation.
- $P \equiv$ employee with Ssn=123456789 works on project x.
- $Q \equiv$ employee e works on project x.

Express the query in tuple relational calculus, using the rules

- $(\forall x)(P(x)) \equiv \textbf{NOT}(\exists x)(\textbf{NOT}(P(x)))$.
- $(\textbf{IF } P \textbf{ THEN } Q) \equiv (\textbf{NOT}(P) \textbf{ OR } Q)$.

8.30. Show how you can specify the following relational algebra operations in both tuple and domain relational calculus.

a. $\sigma_{A=C}(R(A, B, C))$

b. $\pi_{<A, B>}(R(A, B, C))$

c. $R(A, B, C) * S(C, D, E)$

d. $R(A, B, C) \cup S(A, B, C)$

e. $R(A, B, C) \cap S(A, B, C)$

f. $R(A, B, C) = S(A, B, C)$

g. $R(A, B, C) \times S(D, E, F)$

h. $R(A, B) \div S(A)$

8.31. Suggest extensions to the relational calculus so that it may express the following types of operations that were discussed in Section 8.4: (a) aggregate functions and grouping; (b) OUTER JOIN operations; (c) recursive closure queries.

8.32. A nested query is a query within a query. More specifically, a nested query is a parenthesized query whose result can be used as a value in a number of places, such as instead of a relation. Specify the following queries on the database specified in Figure 5.5 using the concept of nested queries and the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database state in Figure 5.6.

a. List the names of all employees who work in the department that has the employee with the highest salary among all employees.

    b. List the names of all employees whose supervisor's supervisor has '888665555' for Ssn.

    c. List the names of employees who make at least $10,000 more than the employee who is paid the least in the company.

8.33. State whether the following conclusions are true or false:

    a. **NOT** $(P(x)$ **OR** $Q(x)) \rightarrow ($**NOT** $(P(x)))$ **AND** $($**NOT** $(Q(x)))$

    b. **NOT** $(\exists x) (P(x)) \rightarrow \forall\, x\, ($**NOT** $(P(x)))$

    c. $(\exists x) (P(x)) \rightarrow \forall\, x\, ((P(x)))$

Laboratory Exercises

8.34. Specify and execute the following queries in relational algebra (RA) using the RA interpreter on the COMPANY database schema in Figure 5.5.

    a. List the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.

    b. List the names of all employees who have a dependent with the same first name as themselves.

    c. List the names of employees who are directly supervised by Franklin Wong.

    d. List the names of employees who work on every project.

    e. List the names of employees who do not work on any project.

    f. List the names and addresses of employees who work on at least one project located in Houston but whose department has no location in Houston.

    g. List the names of department managers who have no dependents.

8.35. Consider the following MAILORDER relational schema describing the data for a mail order company.

    PARTS(Pno, Pname, Qoh, Price, Olevel)
    CUSTOMERS(Cno, Cname, Street, Zip, Phone)
    EMPLOYEES(Eno, Ename, Zip, Hdate)
    ZIP_CODES(Zip, City)
    ORDERS(Ono, Cno, Eno, Received, Shipped)
    ODETAILS(Ono, Pno, Qty)

Qoh stands for *quantity on hand*: the other attribute names are self-explanatory. Specify and execute the following queries using the RA interpreter on the MAILORDER database schema.

    a. Retrieve the names of parts that cost less than $20.00.

    b. Retrieve the names and cities of employees who have taken orders for parts costing more than $50.00.

    c. Retrieve the pairs of customer number values of customers who live in the same ZIP Code.

d. Retrieve the names of customers who have ordered parts from employees living in Wichita.

e. Retrieve the names of customers who have ordered parts costing less than $20.00.

f. Retrieve the names of customers who have not placed an order.

g. Retrieve the names of customers who have placed exactly two orders.

8.36. Consider the following GRADEBOOK relational schema describing the data for a grade book of a particular instructor. (*Note*: The attributes A, B, C, and D of COURSES store grade cutoffs.)

> CATALOG(Cno, Ctitle)
> STUDENTS(Sid, Fname, Lname, Minit)
> COURSES(Term, Sec_no, Cno, A, B, C, D)
> ENROLLS(Sid, Term, Sec_no)

Specify and execute the following queries using the RA interpreter on the GRADEBOOK database schema.

a. Retrieve the names of students enrolled in the Automata class during the fall 2009 term.

b. Retrieve the Sid values of students who have enrolled in CSc226 and CSc227.

c. Retrieve the Sid values of students who have enrolled in CSc226 or CSc227.

d. Retrieve the names of students who have not enrolled in any class.

e. Retrieve the names of students who have enrolled in all courses in the CATALOG table.

8.37. Consider a database that consists of the following relations.

> SUPPLIER(Sno, Sname)
> PART(Pno, Pname)
> PROJECT(Jno, Jname)
> SUPPLY(Sno, Pno, Jno)

The database records information about suppliers, parts, and projects and includes a ternary relationship between suppliers, parts, and projects. This relationship is a many-many-many relationship. Specify and execute the following queries using the RA interpreter.

a. Retrieve the part numbers that are supplied to exactly two projects.

b. Retrieve the names of suppliers who supply more than two parts to project 'J1'.

c. Retrieve the part numbers that are supplied by every supplier.

d. Retrieve the project names that are supplied by supplier 'S1' only.

e. Retrieve the names of suppliers who supply at least two different parts each to at least two different projects.

8.38. Specify and execute the following queries for the database in Exercise 5.16 using the RA interpreter.

    a. Retrieve the names of students who have enrolled in a course that uses a textbook published by Addison-Wesley-Longman.

    b. Retrieve the names of courses in which the textbook has been changed at least once.

    c. Retrieve the names of departments that adopt textbooks published by Addison-Wesley only.

    d. Retrieve the names of departments that adopt textbooks written by Navathe and published by Addison-Wesley.

    e. Retrieve the names of students who have never used a book (in a course) written by Navathe and published by Addison-Wesley.

8.39. Repeat Laboratory Exercises 8.34 through 8.38 in domain relational calculus (DRC) by using the DRC interpreter.

Selected Bibliography

Codd (1970) defined the basic relational algebra. Date (1983a) discusses outer joins. Work on extending relational operations is discussed by Carlis (1986) and Ozsoyoglu et al. (1985). Cammarata et al. (1989) extends the relational model integrity constraints and joins.

Codd (1971) introduced the language Alpha, which is based on concepts of tuple relational calculus. Alpha also includes the notion of aggregate functions, which goes beyond relational calculus. The original formal definition of relational calculus was given by Codd (1972), which also provided an algorithm that transforms any tuple relational calculus expression to relational algebra. The QUEL (Stonebraker et al., 1976) is based on tuple relational calculus, with implicit existential quantifiers, but no universal quantifiers, and was implemented in the INGRES system as a commercially available language. Codd defined relational completeness of a query language to mean at least as powerful as relational calculus. Ullman (1988) describes a formal proof of the equivalence of relational algebra with the safe expressions of tuple and domain relational calculus. Abiteboul et al. (1995) and Atzeni and deAntonellis (1993) give a detailed treatment of formal relational languages.

Although ideas of domain relational calculus were initially proposed in the QBE language (Zloof, 1975), the concept was formally defined by Lacroix and Pirotte (1977a). The experimental version of the Query-By-Example system is described in Zloof (1975). The ILL (Lacroix & Pirotte, 1977b) is based on domain relational calculus. Whang et al. (1990) extends QBE with universal quantifiers. Visual query languages, of which QBE is an example, are being proposed as a means of querying databases; conferences such as the Visual Database Systems Working Conference (e.g., Arisawa & Catarci (2000) or Zhou & Pu (2002)) present a number of proposals for such languages.

Relational Database Design by ER- and EER-to-Relational Mapping

T his chapter discusses how to *design a relational database schema* based on a conceptual schema design. Figure 3.1 presented a high-level view of the database design process. In this chapter we focus on the **logical database design** step of database design, which is also known as **data model mapping**. We present the procedures to create a relational schema from an entity–relationship (ER) or an enhanced ER (EER) schema. Our discussion relates the constructs of the ER and EER models, presented in Chapters 3 and 4, to the constructs of the relational model, presented in Chapters 5 through 8. Many computer-aided software engineering (CASE) tools are based on the ER or EER models, or other similar models, as we have discussed in Chapters 3 and 4. Many tools use ER or EER diagrams or variations to develop the schema graphically and collect information about the data types and constraints, then convert the ER/EER schema automatically into a relational database schema in the DDL of a specific relational DBMS. The design tools employ algorithms similar to the ones presented in this chapter.

We outline a seven-step algorithm in Section 9.1 to convert the basic ER model constructs—entity types (strong and weak), binary relationships (with various structural constraints), *n*-ary relationships, and attributes (simple, composite, and multivalued)—into relations. Then, in Section 9.2, we continue the mapping algorithm by describing how to map EER model constructs—specialization/generalization and union types (categories)—into relations. Section 9.3 summarizes the chapter.

9.1 Relational Database Design Using ER-to-Relational Mapping

9.1.1 ER-to-Relational Mapping Algorithm

In this section we describe the steps of an algorithm for ER-to-relational mapping. We use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 9.1, and the corresponding COMPANY relational database schema is shown in Figure 9.2 to illustrate the

Figure 9.1
The ER conceptual schema diagram for the COMPANY database.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

mapping steps. We assume that the mapping will create tables with simple single-valued attributes. The relational model constraints defined in Chapter 5, which include primary keys, unique keys (if any), and referential integrity constraints on the relations, will also be specified in the mapping results.

Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type *E* in the ER schema, create a relation *R* that includes all the simple attributes of *E*. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of *E* as the primary key for *R*. If the chosen key of *E* is a composite, then the set of simple attributes that form it will together form the primary key of *R*.

If multiple keys were identified for *E* during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify additional (unique) keys of relation *R*. Knowledge about keys is also kept for indexing purposes and other types of analyses.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure 9.2 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT from Figure 9.1. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include

the attributes Super_ssn and Dno of EMPLOYEE, Mgr_ssn and Mgr_start_date of DEPARTMENT, and Dnum of PROJECT. In our example, we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively. Knowledge that Dname of DEPARTMENT and Pname of PROJECT are unique keys is kept for possible use later in the design.

The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple represents an entity instance. The result after this mapping step is shown in Figure 9.3(a).

Step 2: Mapping of Weak Entity Types. For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R. In addition, include as foreign key attributes of R, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any. If there is a weak entity type E_2 whose owner is also a weak entity type E_1, then E_1 should be mapped before E_2 to determine its primary key first.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT (see Figure 9.3(b)). We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type— as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not

---

Figure 9.3
Illustration of some mapping steps.
(a) *Entity* relations after step 1.
(b) Additional *weak entity* relation after step 2.
(c) *Relationship* relations after step 5.
(d) Relation representing multivalued attribute after step 6.

(a) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary

DEPARTMENT

Dname	Dnumber

PROJECT

Pname	Pnumber	Plocation

(b) DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

(c) WORKS_ON

Essn	Pno	Hours

(d) DEPT_LOCATIONS

Dnumber	Dlocation

necessary. The primary key of the DEPENDENT relation is the combination {Essn, Dependent_name}, because Dependent_name (also renamed from Name in Figure 9.1) is the partial key of DEPENDENT.

It is common to choose the propagate (CASCADE) option for the referential triggered action (see Section 6.2) on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both ON UPDATE and ON DELETE.

Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the cross-reference or relationship relation approach. The first approach is the most useful and should be followed unless special conditions exist, as we discuss below.

1. **Foreign key approach:** Choose one of the relations—S, say—and include as a foreign key in S the primary key of T. It is better to choose an entity type with *total participation* in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.

   In our example, we map the 1:1 relationship type MANAGES from Figure 9.1 by choosing the participating entity type DEPARTMENT to serve in the role of S because its participation in the MANAGES relationship type is total (every department has a manager). We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it to Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date (see Figure 9.2).

   Note that it is possible to include the primary key of S as a foreign key in T instead. In our example, this amounts to having a foreign key attribute, say Department_managed in the EMPLOYEE relation, but it will have a NULL value for employee tuples who do not manage a department. This would be a bad choice, because if only 2% of employees manage a department, then 98% of the foreign keys would be NULL in this case. Another possibility is to have foreign keys in both relations S and T redundantly, but this creates redundancy and incurs a penalty for consistency maintenance.

2. **Merged relation approach:** An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when *both participations are total,* as this would indicate that the two tables will have the exact same number of tuples at all times.

3. **Cross-reference or relationship relation approach:** The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation R is called a **relationship relation** (or sometimes a **lookup table**), because each

tuple in R represents a relationship instance that relates one tuple from S with one tuple from T. The relation R will include the primary key attributes of S and T as foreign keys to S and T. The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R. The drawback is having an extra relation, and requiring extra join operations when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types. There are two possible approaches: (1) the foreign key approach and (2) the cross-reference or relationship relation approach. The first approach is generally preferred as it reduces the number of tables.

1. **The foreign key approach:** For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the *N-side* of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S.

   To apply this approach to our example, we map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 9.1. For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn. The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation. These foreign keys are shown in Figure 9.2.

2. **The relationship relation approach:** An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation R whose attributes are the primary keys of S and T, which will also be foreign keys to S and T. The primary key of R is the same as the primary key of S. This option can be used if few tuples in S participate in the relationship to avoid excessive NULL values in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types. In the traditional relational model with no multivalued attributes, the only option for M:N relationships is the **relationship relation (cross-reference) option**. For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their *combination* will form the primary key of S. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for

1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate *relationship relation S*.

In our example, we map the M:N relationship type WORKS_ON from Figure 9.1 by creating the relation WORKS_ON in Figure 9.2. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively (renaming is *not required;* it is a design choice). We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}. This **relationship relation** is shown in Figure 9.3(c).

The propagate (CASCADE) option for the referential triggered action (see Section 4.2) should be specified on the foreign keys in the relation corresponding to the relationship *R*, since each relationship instance has an existence dependency on each of the entities it relates. This can be used for both ON UPDATE and ON DELETE.

Although we can map 1:1 or 1:N relationships in a manner similar to M:N relationships by using the cross-reference (relationship relation) approach, as we discussed earlier, this is only recommended when few relationship instances exist, in order to avoid NULL values in foreign keys. In this case, the primary key of the relationship relation will be *only one* of the foreign keys that reference the participating entity relations. For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side. For a 1:1 relationship, either foreign key can be used as the primary key of the relationship relation.

Step 6: Mapping of Multivalued Attributes. For each multivalued attribute *A*, create a new relation *R*. This relation *R* will include an attribute corresponding to *A*, plus the primary key attribute *K*—as a foreign key in *R*—of the relation that represents the entity type or relationship type that has *A* as a multivalued attribute. The primary key of *R* is the combination of *A* and *K*. If the multivalued attribute is composite, we include its simple components.

In our example, we create a relation DEPT_LOCATIONS (see Figure 9.3(d)). The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, whereas Dnumber—as foreign key—represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}. A separate tuple will exist in DEPT_LOCATIONS for each location that a department has. It is important to note that in more recent versions of the relational model that allow array data types, the multivalued attribute can be mapped to an array attribute rather than requiring a separate table.

The propagate (CASCADE) option for the referential triggered action (see Section 6.2) should be specified on the foreign key in the relation *R* corresponding to the multivalued attribute for both ON UPDATE and ON DELETE. We should also note that the key of *R* when mapping a composite, multivalued attribute requires some analysis of the meaning of the component attributes. In some cases, when a multivalued attribute is composite, only some of the component attributes are required

to be part of the key of *R*; these attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute (see Section 3.5).

Figure 9.2 shows the COMPANY relational database schema obtained with steps 1 through 6, and Figure 5.6 shows a sample database state. Notice that we did not yet discuss the mapping of *n*-ary relationship types (*n* > 2) because none exist in Figure 9.1 ; these are mapped in a similar way to M:N relationship types by including the following additional step in the mapping algorithm.

Step 7: Mapping of *N*-ary Relationship Types. We use the **relationship relation option**. For each *n*-ary relationship type *R*, where *n* > 2, create a new relationship relation *S* to represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the *n*-ary relationship type (or simple components of composite attributes) as attributes of *S*. The primary key of *S* is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types *E* participating in *R* is 1, then the primary key of *S* should not include the foreign key attribute that references the relation *E′* corresponding to *E* (see the discussion in Section 3.9.2 concerning constraints on *n*-ary relationships).

Consider the ternary relationship type SUPPLY in Figure 3.17, which relates a SUPPLIER *s*, PART *p*, and PROJECT *j* whenever *s* is currently supplying *p* to *j*; this can be mapped to the relation SUPPLY shown in Figure 9.4, whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

9.1.2 Discussion and Summary of Mapping for ER Model Constructs

Table 9.1 summarizes the correspondences between ER and relational model constructs and constraints.

Figure 9.4

Mapping the *n*-ary relationship type SUPPLY from Figure 3.17(a).

Table 9.1   Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

One of the main points to note in a relational schema, in contrast to an ER schema, is that relationship types are not represented explicitly; instead, they are represented by having two attributes A and B, one a primary key and the other a foreign key (over the same domain) included in two relations S and T. Two tuples in S and T are related when they have the same value for A and B. By using the EQUIJOIN operation (or NATURAL JOIN if the two join attributes have the same name) over $S.A$ and $T.B$, we can combine all pairs of related tuples from S and T and materialize the relationship. When a binary 1:1 or 1:N relationship type is involved and the foreign key mapping is used, a single join operation is usually needed. When the relationship relation approach is used, such as for a binary M:N relationship type, two join operations are needed, whereas for n-ary relationship types, n joins are needed to fully materialize the relationship instances.

For example, to form a relation that includes the employee name, project name, and hours that the employee works on each project, we need to connect each EMPLOYEE tuple to the related PROJECT tuples via the WORKS_ON relation in Figure 9.2. Hence, we must apply the EQUIJOIN operation to the EMPLOYEE and WORKS_ON relations with the join condition EMPLOYEE.Ssn = WORKS_ON.Essn, and then apply another EQUIJOIN operation to the resulting relation and the PROJECT relation with join condition WORKS_ON.Pno = PROJECT.Pnumber. In general, when multiple relationships need to be traversed, numerous join operations must be specified. The user must always be aware of the foreign key attributes in order to use them correctly in combining related tuples from two or more relations. This is sometimes considered to be a drawback of the relational data model, because the foreign key/primary key correspondences are not always obvious upon inspection of relational schemas. If an EQUIJOIN is performed among attributes of two relations that do not represent a foreign key/primary key relationship, the result can often be meaningless and may lead to spurious data. For example, the reader can try joining the PROJECT and DEPT_LOCATIONS relations on the condition Dlocation = Plocation and examine the result.

In the relational schema we create a separate relation for *each* multivalued attribute. For a particular entity with a set of values for the multivalued attribute, the key attribute value of the entity is repeated once for each value of the multivalued attribute in a separate tuple because the basic relational model does *not* allow multiple values (a list, or a set of values) for an attribute in a single tuple. For example, because department 5 has three locations, three tuples exist in the DEPT_LOCATIONS relation in Figure 3.6; each tuple specifies one of the locations. In our example, we apply EQUIJOIN to DEPT_LOCATIONS and DEPARTMENT on the Dnumber attribute to get the values of all locations along with other DEPARTMENT attributes. In the resulting relation, the values of the other DEPARTMENT attributes are repeated in separate tuples for every location that a department has.

The basic relational algebra does not have a NEST or COMPRESS operation that would produce a set of tuples of the form {<'1', 'Houston'>, <'4', 'Stafford'>, <'5', {'Bellaire', 'Sugarland', 'Houston'}>} from the DEPT_LOCATIONS relation in Figure 3.6. This is a serious drawback of the basic normalized or *flat* version of the relational model. The object data model and object-relational systems (see Chapter 12) do allow multivalued attributes by using the array type for the attribute.

9.2 Mapping EER Model Constructs to Relations

In this section, we discuss the mapping of EER model constructs to relations by extending the ER-to-relational mapping algorithm that was presented in Section 9.1.1.

9.2.1 Mapping of Specialization or Generalization

There are several options for mapping a number of subclasses that together form a specialization (or alternatively, that are generalized into a superclass), such as the {SECRETARY, TECHNICIAN, ENGINEER} subclasses of EMPLOYEE in Figure 4.4. The two main options are to map the whole specialization into a **single table**, or to map it into **multiple tables**. Within each option are variations that depend on the constraints on the specialization/generalization.

We can add a further step to our ER-to-relational mapping algorithm from Section 9.1.1, which has seven steps, to handle the mapping of specialization. Step 8, which follows, gives the most common options; other mappings are also possible. We discuss the conditions under which each option should be used. We use Attrs(R) to denote *the attributes of a relation R*, and PK(R) to denote the *primary key of R*. First we describe the mapping formally, then we illustrate it with examples.

Step 8: Options for Mapping Specialization or Generalization. Convert each specialization with m subclasses $\{S_1, S_2, \ldots, S_m\}$ and (generalized) superclass C, where the attributes of C are $\{k, a_1, \ldots, a_n\}$ and k is the (primary) key, into relation schemas using one of the following options:

- **Option 8A: Multiple relations—superclass and subclasses.** Create a relation L for C with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\}$ and PK(L) = k. Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes Attrs(L_i) = $\{k\} \cup \{$attributes of $S_i\}$ and PK(L_i) = k. This option works for any specialization (total or partial, disjoint or overlapping).

- **Option 8B: Multiple relations—subclass relations only.** Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes Attrs(L_i) = $\{$attributes of $S_i\} \cup \{k, a_1, \ldots, a_n\}$ and PK(L_i) = k. This option only works for a specialization whose subclasses are *total* (every entity in the superclass must belong to (at least) one of the subclasses). Additionally, it is only recommended if the specialization has the *disjointedness constraint* (see Section 4.3.1). If the specialization is *overlapping*, the same entity may be duplicated in several relations.

- **Option 8C: Single relation with one type attribute.** Create a single relation L with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\} \cup \{$attributes of $S_1\} \cup \ldots \cup \{$attributes of $S_m\} \cup \{t\}$ and PK(L) = k. The attribute t is called a **type** (or **discriminating**) attribute whose value indicates the subclass to which each tuple belongs, if any. This option works only for a specialization whose subclasses are *disjoint,* and has the potential for generating many NULL values if many specific (local) attributes exist in the subclasses.

- **Option 8D: Single relation with multiple type attributes.** Create a single relation schema L with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\} \cup \{$attributes of $S_1\} \cup \ldots \cup \{$attributes of $S_m\} \cup \{t_1, t_2, \ldots, t_m\}$ and PK(L) = k. Each t_i, $1 \leq i \leq m$, is a **Boolean type attribute** indicating whether or not a tuple belongs to subclass S_i. This option is used for a specialization whose subclasses are *overlapping* (but will also work for a disjoint specialization).

Options 8A and 8B are the **multiple-relation options**, whereas options 8C and 8D are the **single-relation options**. Option 8A creates a relation L for the superclass C and its attributes, plus a relation L_i for each subclass S_i; each L_i includes the specific (local) attributes of S_i, plus the primary key of the superclass C, which is propagated to L_i and becomes its primary key. It also becomes a foreign key to the superclass relation. An EQUIJOIN operation on the primary key between any L_i and L produces all the specific and inherited attributes of the entities in S_i. This option is illustrated in Figure 9.5(a) for the EER schema in Figure 4.4. Option 8A works for any constraints on the specialization: disjoint or overlapping, total or partial. Notice that the constraint

$$\pi_{<k>}(L_i) \subseteq \pi_{<k>}(L)$$

must hold for each L_i. This specifies a foreign key from each L_i to L.

In option 8B, the EQUIJOIN operation between each subclass and the superclass is *built into* the schema and the superclass relation L is done away with, as illustrated in Figure 9.5(b) for the EER specialization in Figure 4.3(b). This option works well only when *both* the disjoint and total constraints hold. If the specialization is not total, an entity that does not belong to any of the subclasses S_i is lost. If the specialization is not disjoint, an entity belonging to more than one subclass will have its

(a) EMPLOYEE

Ssn	Fname	Minit	Lname	Birth_date	Address	Job_type

SECRETARY

Ssn	Typing_speed

TECHNICIAN

Ssn	Tgrade

ENGINEER

Ssn	Eng_type

(b) CAR

Vehicle_id	License_plate_no	Price	Max_speed	No_of_passengers

TRUCK

Vehicle_id	License_plate_no	Price	No_of_axles	Tonnage

(c) EMPLOYEE

Ssn	Fname	Minit	Lname	Birth_date	Address	Job_type	Typing_speed	Tgrade	Eng_type

(d) PART

Part_no	Description	Mflag	Drawing_no	Manufacture_date	Batch_no	Pflag	Supplier_name	List_price

Figure 9.5
Options for mapping specialization or generalization. (a) Mapping the EER schema in Figure 4.4 using option 8A.
(b) Mapping the EER schema in Figure 4.3(b) using option 8B. (c) Mapping the EER schema in Figure 4.4 using
option 8C. (d) Mapping Figure 4.5 using option 8D with Boolean type fields Mflag and Pflag.

inherited attributes from the superclass C stored redundantly in more than one table L_i.
With option 8B, no relation holds all the entities in the superclass C; consequently, we
must apply an OUTER UNION (or FULL OUTER JOIN) operation (see Section 6.4) to the L_i
relations to retrieve all the entities in C. The result of the outer union will be similar to
the relations under options 8C and 8D except that the type fields will be missing. When-
ever we search for an arbitrary entity in C, we must search all the m relations L_i.

Options 8C and 8D create a single relation to represent the superclass C and all its
subclasses. An entity that does not belong to some of the subclasses will have NULL
values for the specific (local) attributes of these subclasses. These options are not
recommended if many specific attributes are defined for the subclasses. If few local
subclass attributes exist, however, these mappings are preferable to options 8A
and 8B because they do away with the need to specify JOIN operations; therefore,
they can yield a more efficient implementation for queries.

Option 8C is used to handle disjoint subclasses by including a single **type** (or **image**
or **discriminating**) **attribute** t to indicate to which of the m subclasses each tuple
belongs; hence, the domain of t could be {1, 2, … , m}. If the specialization is partial, t
can have NULL values in tuples that do not belong to any subclass. If the specialization
is attribute-defined, that attribute itself serves the purpose of t and t is not needed; this
option is illustrated in Figure 9.5(c) for the EER specialization in Figure 4.4.

Option 8D is designed to handle overlapping subclasses by including m *Boolean*
type (or **flag**) fields, one for *each* subclass. It can also be used for disjoint subclasses.

Each type field t_i can have a domain {yes, no}, where a value of yes indicates that the tuple is a member of subclass S_i. If we use this option for the EER specialization in Figure 4.4, we would include three type attributes—Is_a_secretary, Is_a_engineer, and Is_a_technician—instead of the Job_type attribute in Figure 9.5(c). Figure 9.5(d) shows the mapping of the specialization from Figure 4.5 using option 8D.

For a multilevel specialization (or generalization) hierarchy or lattice, we do not have to follow the same mapping option for all the specializations. Instead, we can use one mapping option for part of the hierarchy or lattice and other options for other parts. Figure 9.6 shows one possible mapping into relations for the EER lattice in Figure 4.6. Here we used option 8A for PERSON/{EMPLOYEE, ALUMNUS, STUDENT}, and option 8C for EMPLOYEE/{STAFF, FACULTY, STUDENT_ASSISTANT} by including the type attribute Employee_type. We then used the single-table option 8D for STUDENT_ASSISTANT/{RESEARCH_ASSISTANT, TEACHING_ASSISTANT} by including the type attributes Ta_flag and Ra_flag in EMPLOYEE. We also used option 8D for STUDENT/STUDENT_ASSISTANT by including the type attributes Student_assist_flag in STUDENT, and for STUDENT/{GRADUATE_STUDENT, UNDERGRADUATE_STUDENT} by including the type attributes Grad_flag and Undergrad_flag in STUDENT. In Figure 9.6, all attributes whose names end with *type* or *flag* are type fields.

9.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

A shared subclass, such as ENGINEERING_MANAGER in Figure 4.6, is a subclass of several superclasses, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category (union type) as we discussed in Section 4.4. We can apply any of the options discussed in step 8 to a shared subclass, subject to the restrictions discussed in step 8 of the mapping algorithm. In Figure 9.6, options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT. Option 8C is used in the EMPLOYEE relation (Employee_type attribute) and option 8D is used in the STUDENT relation (Student_assist_flag attribute).

Figure 9.6
Mapping the EER specialization lattice in Figure 4.8 using multiple options.

9.2.3 Mapping of Categories (Union Types)

We add another step to the mapping procedure—step 9—to handle categories. A category (or union type) is a subclass of the *union* of two or more superclasses that can have different keys because they can be of different entity types (see Section 4.4). An example is the OWNER category shown in Figure 4.8, which is a subset of the union of three entity types PERSON, BANK, and COMPANY. The other category in that figure, REGISTERED_VEHICLE, has two superclasses that have the same key attribute.

Step 9: Mapping of Union Types (Categories). For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a **surrogate key**, when creating a relation to correspond to the union type. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the relation. In our example in Figure 4.8, we create a relation OWNER to correspond to the OWNER category, as illustrated in Figure 9.7, and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called Owner_id. We also

Figure 9.7

Mapping the EER categories (union types) in Figure 4.8 to relations.

include the surrogate key attribute Owner_id as foreign key in each relation corresponding to a superclass of the category, to specify the correspondence in values between the surrogate key and the original key of each superclass. Notice that if a particular PERSON (or BANK or COMPANY) entity is not a member of OWNER, it would have a NULL value for its Owner_id attribute in its corresponding tuple in the PERSON (or BANK or COMPANY) relation, and it would not have a tuple in the OWNER relation. It is also recommended to add a type attribute (not shown in Figure 9.7) to the OWNER relation to indicate the particular entity type to which each tuple belongs (PERSON or BANK or COMPANY).

For a category whose superclasses have the same key, such as VEHICLE in Figure 4.8, there is no need for a surrogate key. The mapping of the REGISTERED_VEHICLE category, which illustrates this case, is also shown in Figure 9.7.

9.3 Summary

In Section 9.1, we showed how a conceptual schema design in the ER model can be mapped to a relational database schema. An algorithm for ER-to-relational mapping was given and illustrated by examples from the COMPANY database. Table 9.1 summarized the correspondences between the ER and relational model constructs and constraints. Next, we added additional steps to the algorithm in Section 9.2 for mapping the constructs from the EER model into the relational model. Similar algorithms are incorporated into graphical database design tools to create a relational schema from a conceptual schema design automatically.

Review Questions

9.1. **(a)** Discuss the correspondences between the ER model constructs and the relational model constructs. Show how each ER model construct can be mapped to the relational model and discuss any alternative mappings. **(b)** Discuss the options for mapping EER model constructs to relations, and the conditions under which each option could be used.

Exercises

9.2. Map the UNIVERSITY database schema shown in Figure 3.20 into a relational database schema.

9.3. Try to map the relational schema in Figure 6.14 into an ER schema. This is part of a process known as *reverse engineering*, where a conceptual schema is created for an existing implemented database. State any assumptions you make.

Figure 9.8
An ER schema for a SHIP_TRACKING database.

9.4. Figure 9.8 shows an ER schema for a database that can be used to keep track of transport ships and their locations for maritime authorities. Map this schema into a relational schema and specify all primary keys and foreign keys.

9.5. Map the BANK ER schema of Exercise 3.23 (shown in Figure 3.21) into a relational schema. Specify all primary keys and foreign keys. Repeat for the AIRLINE schema (Figure 3.20) of Exercise 3.19 and for the other schemas for Exercises 3.16 through 3.24.

9.6. Map the EER diagrams in Figures 4.9 and 4.12 into relational schemas. Justify your choice of mapping options.

9.7. Is it possible to successfully map a binary M:N relationship type without requiring a new relation? Why or why not?

Figure 9.9
EER diagram for
a car dealer.

9.8. Consider the EER diagram in Figure 9.9 for a car dealer.

Map the EER schema into a set of relations. For the VEHICLE to CAR/TRUCK/SUV generalization, consider the four options presented in Section 9.2.1 and show the relational schema design under each of those options.

9.9. Using the attributes you provided for the EER diagram in Exercise 4.27, map the complete schema into a set of relations. Choose an appropriate option out of 8A thru 8D from Section 9.2.1 in doing the mapping of generalizations and defend your choice.

Laboratory Exercises

9.10. Consider the ER design for the UNIVERSITY database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 3.31. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

9.11. Consider the ER design for the MAIL_ORDER database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 3.32. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

9.12. Consider the ER design for the CONFERENCE_REVIEW database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 3.34. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

9.13. Consider the EER design for the GRADE_BOOK database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 4.28. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

9.14. Consider the EER design for the ONLINE_AUCTION database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 4.29. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

Selected Bibliography

The original ER-to-relational mapping algorithm was described in Chen's classic paper (Chen, 1976). Batini et al. (1992) discuss a variety of mapping algorithms from ER and EER models to legacy models and vice versa.

part 4

Database Programming Techniques

This page intentionally left blank

chapter **10**

Introduction to SQL
Programming Techniques

In Chapters 6 and 7, we described several aspects of the SQL language, which is the standard for relational databases. We described the SQL statements for data definition, schema modification, queries, views, and updates. We also described how various constraints on the database contents, such as key and referential integrity constraints, are specified.

In this chapter and the next, we discuss some of the methods that have been developed for accessing databases from programs. Most database access in practical applications is accomplished through software programs that implement **database applications**. This software is usually developed in a general-purpose programming language such as Java, C/C++/C#, COBOL (historically), or some other programming language. In addition, many scripting languages, such as PHP, Python, and JavaScript, are also being used for programming of database access within Web applications. In this chapter, we focus on how databases can be accessed from the traditional programming languages C/C++ and Java, whereas in the next chapter we introduce how databases are accessed from scripting languages such as PHP. Recall from Section 2.3.1 that when database statements are included in a program, the general-purpose programming language is called the *host language*, whereas the database language—SQL, in our case—is called the *data sublanguage*. In some cases, special *database programming languages* are developed specifically for writing database applications. Although many of these were developed as research prototypes, some notable database programming languages have widespread use, such as Oracle's PL/SQL (Programming Language/SQL).

It is important to note that database programming is a very broad topic. There are whole textbooks devoted to each database programming technique and how that technique is realized in a specific system. New techniques are developed all the

time, and changes to existing techniques are incorporated into newer system versions and languages. An additional difficulty in presenting this topic is that although there are SQL standards, these standards themselves are continually evolving, and each DBMS vendor may have some variations from the standard. Because of this, we have chosen to give an introduction to some of the main types of database programming techniques and to compare these techniques, rather than study one particular method or system in detail. The examples we give serve to illustrate the main differences that a programmer would face when using each of these database programming techniques. We will try to use the SQL standards in our examples rather than describe a specific system. When using a specific system, the materials in this chapter can serve as an introduction, but should be augmented with the system manuals or with books describing the specific system.

We start our presentation of database programming in Section 10.1 with an overview of the different techniques developed for accessing a database from programs. Then, in Section 10.2, we discuss the rules for embedding SQL statements into a general-purpose programming language, generally known as *embedded SQL*. This section also briefly discusses *dynamic SQL*, in which queries can be dynamically constructed at runtime, and presents the basics of the SQLJ variation of embedded SQL that was developed specifically for the programming language Java. In Section 10.3, we discuss the technique known as *SQL/CLI* (Call Level Interface), in which a library of procedures and functions is provided for accessing the database. Various sets of library functions have been proposed. The SQL/CLI set of functions is the one given in the SQL standard. Another widely used library of functions is *ODBC* (Open Data Base Connectivity), which has many similarities to SQL/CLI; in fact, SQL/CLI can be thought of as the standardized version of ODBC. A third library of classes—which we do describe—is *JDBC*; this was developed specifically for accessing databases from the Java object-oriented programming language (OOPL). In OOPL, a library of classes is used instead of a library of functions and procedures, and each class has its own operations and functions. In Section 10.4 we discuss *SQL/PSM* (Persistent Stored Modules), which is a part of the SQL standard that allows program modules—procedures and functions—to be stored by the DBMS and accessed through SQL; this also specifies a procedural *database programming language* for writing the persistent stored modules. We briefly compare the three approaches to database programming in Section 10.5, and provide a chapter summary in Section 10.6.

10.1 Overview of Database Programming Techniques and Issues

We now turn our attention to the techniques that have been developed for accessing databases from programs and, in particular, to the issue of how to access SQL databases from application programs. Our presentation of SQL in Chapters 6 and 7 focused on the language constructs for various database operations—from schema definition and constraint specification to querying, updating, and specifying views.

Most database systems have an **interactive interface** where these SQL commands can be typed directly into a monitor for execution by the database system. For example, in a computer system where the Oracle RDBMS is installed, the command `SQLPLUS` starts the interactive interface. The user can type SQL commands or queries directly over several lines, ended by a semicolon and the Enter key (that is, `";<cr>"`). Alternatively, a **file of commands** can be created and executed through the interactive interface by typing `@<`*filename*`>`. The system will execute the commands written in the file and display the results, if any.

The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, in practice, the majority of database interactions are executed through programs that have been carefully designed and tested. These programs are generally known as **application programs** or **database applications**, and are used as *canned transactions* by the end users, as discussed in Section 1.4.3. Another common use of database programming is to access a database through an application program that implements a **Web interface**, for example, when making airline reservations or online purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands. Chapter 11 gives an overview of Web database programming using PHP, a scripting language that has recently become widely used.

In this section, first we give an overview of the main approaches to database programming. Then we discuss some of the problems that occur when trying to access a database from a general-purpose programming language, and the typical sequence of commands for interacting with a database from a software program.

10.1.1 Approaches to Database Programming

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

1. **Embedding database commands in a general-purpose programming language.** In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string `EXEC SQL`, which precedes all SQL commands in a host language program.[1] A **precompiler** or **preproccessor** scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code. This technique is generally referred to as **embedded SQL**.

2. **Using a library of database functions or classes.** A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, prepare a query, execute a query, execute an update, loop over the query result on record at a time, and so on. The actual database query and update commands and any

---

[1]Other prefixes are sometimes used, but this is the most common.

other necessary information are included as parameters in the function calls. This approach provides what is known as an **application programming interface** (**API**) for accessing a database from application programs. For object-oriented programming languages (OOPLs), a **class library** is used. For example, Java has the JDBC class library, which can generate various types of objects such as: connection objects to a particular database, query objects, and query result objects. Each type of object has a set of operations associated with the class corresponding to the object.

3. **Designing a brand-new language.** A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full-fledged programming language. An example of this approach is Oracle's PL/SQL. The SQL standard has the SQL/PSM language for specifying stored procedures.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some database access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is *impedance mismatch*, which does not occur in the third approach.

10.1.2 Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: columns (attributes) and their data types, rows (also referred to as tuples or records), and tables (sets or multisets of records). The first problem that may occur is that the *data types of the programming language* differ from the *attribute data types* that are available in the data model. Hence, it is necessary to have a **binding** for each host programming language that specifies for each attribute type the compatible programming language types. A different binding is needed *for each programming language* because different languages have different data types. For example, the data types available in C/C++ and Java are different, and both differ from the SQL data types, which are the standard data types for relational databases.

Another problem occurs because the results of most queries are sets or multisets of tuples (rows), and each tuple is formed of a sequence of attribute values. In the program, it is often necessary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the *query result data structure*, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a **query result** in order to access a single tuple at a time and to extract individual values from the tuple. The extracted attribute values are typically copied to appropriate program variables for further processing by the program. A **cursor** or **iterator variable** is typically used to loop over the tuples in a query result. Individual values within each tuple are then extracted into distinct program variables of the appropriate type.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is Oracle's PL/SQL. The SQL standard also has a proposal for such a database programming language, known as *SQL/PSM*. For object databases, the object data model (see Chapter 12) is quite similar to the data model of the Java programming language, so the impedance mismatch is greatly reduced when Java is used as the host language for accessing a Java-compatible object database. Several database programming languages have been implemented as research prototypes (see the Selected Bibliography).

10.1.3 Typical Sequence of Interaction in Database Programming

When a programmer or software engineer writes a program that requires access to a database, it is quite common for the program to be running on one computer system while the database is installed on another. Recall from Section 2.5 that a common architecture for database access is the three-tier client/server model, where a top-tier **client program** handles display of information on a laptop or mobile device usually as a Web client or mobile app, a middle-tier **application program** implements the logic of a business software application but includes some calls to one or more **database servers** at the bottom tier to access or update the data.[2] When writing such an application program, a common sequence of interaction is the following:

1. When the application program requires access to a particular database, the program must first *establish* or *open* a **connection** to the database server. Typically, this involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.

2. Once the connection is established, the program can interact with the database by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.

3. When the program no longer needs access to a particular database, it should *terminate* or *close* the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established simultaneously.

In the next three sections, we discuss examples of each of the three main approaches to database programming. Section 10.2 describes how SQL is *embedded* into a programming language. Section 10.3 discusses how *function calls* and *class libraries* are used to access the database using SQL/CLI (similar to ODBC) and JDBC, and Section 10.4 discusses an extension to SQL called SQL/PSM that allows *general-purpose*

---

[2]As we discussed in Section 2.5, there are two-tier and three-tier architectures; to keep our discussion simple, we will assume a two-tier client/server architecture here.

programming constructs for defining modules (procedures and functions) that are stored within the database system.[3] Section 10.5 compares these approaches.

10.2 Embedded SQL, Dynamic SQL, and SQLJ

In this section, we give an overview of the techniques for embedding SQL statements in a general-purpose programming language. We focus on two languages: C and Java. The examples used with the C language, known as **embedded SQL**, are presented in Sections 10.2.1 through 10.2.3, and can be adapted to other similar programming languages. The examples using Java, known as **SQLJ**, are presented in Sections 10.2.4 and 10.2.5. In this embedded approach, the programming language is called the **host language**. Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program.

10.2.1 Retrieving Single Tuples with Embedded SQL

To illustrate the concepts of embedded SQL, we will use C as the host programming language.[4] In a C program, an embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a **preprocessor** (or **precompiler**) can separate embedded SQL statements from the host language source code. The SQL statements within a program are terminated by a matching END-EXEC or by a semicolon (;). Similar rules apply to embedding SQL in other programming languages.

Within an embedded SQL command, the programmer can refer to specially declared C program variables; these are called **shared variables** because they are used in both the C program and the embedded SQL statements. Shared variables are prefixed by a colon (:) *when they appear in an SQL statement*. This distinguishes program variable names from the names of database schema constructs such as attributes (column names) and relations (table names). It also allows program variables to have the same names as attribute names, since they are distinguishable by the colon (:) prefix in the SQL statement. Names of database schema constructs—such as attributes and relations—can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the colon (:) prefix.

Suppose that we want to write C programs to process the COMPANY database in Figure 5.5. We need to declare program variables to match the types of the database attributes that the program will process. The programmer can choose the names of the **program variables**; they may or may not have names that are identical to their

---

[3]SQL/PSM illustrates how typical general-purpose programming language constructs—such as loops and conditional structures—can be incorporated into SQL.

[4]Our discussion here also applies to the C++ or C# programming languages, since we do not use any of the object-oriented features, but focus on the database programming mechanism.

```
0) int loop ;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar dname [16], fname [16], lname [16], address [31] ;
3) char ssn [10], bdate [11], sex [2], minit [2] ;
4) float salary, raise ;
5) int dno, dnumber ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

Figure 10.1
C program variables used in the embedded SQL examples E1 and E2.

corresponding database attributes. We will use the C program variables declared in Figure 10.1 for all our examples and show C program segments *without variable declarations*. Shared variables are declared within a **declare section** in the program, as shown in Figure 10.1 (lines 1 through 7).[5] A few of the common bindings of C types to SQL types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C data types `long`, `short`, `float`, and `double`, respectively. Fixed-length and varying-length strings (CHAR [*i*], VARCHAR [*i*]) in SQL can be mapped to arrays of characters (`char [i+1]`, `varchar [i+1]`) in C that are one character longer than the SQL type because strings in C are terminated by a NULL character (`\0`), which is not part of the character string itself.[6] Although `varchar` is not a standard C data type, it is permitted when C is used for SQL database programming.

Notice that the only embedded SQL commands in Figure 10.1 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements—as long as they are preceded by a colon (:). Lines 2 through 5 are regular C program declarations. The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables from the COMPANY database in Figure 5.5 that was declared by the SQL DDL in Figure 6.1. The variables declared in line 6—SQLCODE and SQLSTATE—are called **SQL communication variables;** they are used to communicate errors and exception conditions between the database system and the executing program. Line 0 shows a program variable *loop* that will not be used in any embedded SQL statement, so it is declared outside the SQL declare section.

Connecting to the Database. The SQL command for establishing a connection to a database has the following form:

   CONNECT TO <server name>**AS** <connection name>
   AUTHORIZATION <user account name and password> ;

In general, since a user or program can access several database servers, several connections can be established, but only one connection can be active at any point in

---

[5]We use line numbers in our code segments for easy reference; these numbers are not part of the actual code.

[6]SQL strings can also be mapped to char* types in C.

time. The programmer or user can use the <connection name> to change from the currently active connection to a different one by using the following command:

    SET CONNECTION <connection name> ;

Once a connection is no longer needed, it can be terminated by the following command:

    DISCONNECT <connection name> ;

In the examples in this chapter, we assume that the appropriate connection has already been established to the COMPANY database, and that it is the currently active connection.

Communication variables SQLCODE and SQLSTATE. The two special **communication variables** that are used by the DBMS to communicate exception or error conditions to the program are SQLCODE and SQLSTATE. The **SQLCODE** variable shown in Figure 10.1 is an integer variable. After each database command is executed, the DBMS returns a value in SQLCODE. A value of 0 indicates that the statement was executed successfully by the DBMS. If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result. If SQLCODE < 0, this indicates some error has occurred. In some systems—for example, in the Oracle RDBMS—SQLCODE is a field in a record structure called SQLCA (SQL communication area), so it is referenced as SQLCA.SQLCODE. In this case, the definition of SQLCA must be included in the C program by including the following line:

    EXEC SQL include SQLCA ;

In later versions of the SQL standard, a communication variable called **SQLSTATE** was added, which is a string of five characters. A value of '00000' in SQLSTATE indicates no error or exception; other values indicate various errors or exceptions. For example, '02000' indicates 'no more data' when using SQLSTATE. Currently, both SQLSTATE and SQLCODE are available in the SQL standard. Many of the error and exception codes returned in SQLSTATE are supposed to be standardized for all SQL vendors and platforms,[7] whereas the codes returned in SQLCODE are not standardized but are defined by the DBMS vendor. Hence, it is generally better to use SQLSTATE because this makes error handling in the application programs independent of a particular DBMS. As an exercise, the reader should rewrite the examples given later in this chapter using SQLSTATE instead of SQLCODE.

Example of Embedded SQL Programming. Our first example to illustrate embedded SQL programming is a repeating program segment (loop) that takes as input a Social Security number of an employee and prints some information from the corresponding EMPLOYEE record in the database. The C program code is shown as program segment E1 in Figure 10.2. The program reads (inputs) an Ssn value

---

[7]In particular, SQLSTATE codes starting with the characters 0 through 4 or A through H are supposed to be standardized, whereas other values can be implementation-defined.

```
    //Program Segment E1:
0)  loop = 1 ;
1)  while (loop) {
2)    prompt("Enter a Social Security Number: ", ssn) ;
3)    EXEC SQL
4)      SELECT Fname, Minit, Lname, Address, Salary
5)      INTO :fname, :minit, :lname, :address, :salary
6)      FROM EMPLOYEE WHERE Ssn = :ssn ;
7)    if (SQLCODE = = 0) printf(fname, minit, lname, address, salary)
8)      else printf("Social Security Number does not exist: ", ssn) ;
9)    prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)   }
```

Figure 10.2
Program segment E1,
a C program segment
with embedded SQL.

and then retrieves the EMPLOYEE tuple with that Ssn from the database via the embedded SQL command. The **INTO** clause (line 5) specifies the program variables into which attribute values from the database record are retrieved. C program variables in the INTO clause are prefixed with a colon (:), as we discussed earlier. The INTO clause can be used in this manner only when the query result is a *single record*; if multiple records are retrieved, an error will be generated. We will see how multiple records are handled in Section 10.2.2.

Line 7 in E1 illustrates the communication between the database and the program through the special variable SQLCODE. If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions. Line 7 checks this and assumes that if an error occurred, it was because no EMPLOYEE tuple existed with the given Ssn; therefore it outputs a message to that effect (line 8).

When a single record is retrieved as in example E1, the programmer can assign its attribute values directly to C program variables in the INTO clause, as in line 5. In general, an SQL query can retrieve many tuples. In that case, the C program will typically loop through the retrieved tuples and process them one at a time. The concept of a *cursor* is used to allow tuple-at-a-time processing of a query result by the host language program. We describe cursors next.

10.2.2 Processing Query Results Using Cursors

A **cursor** is a variable that refers to a *single tuple* (*row*) from a **query result** that retrieves a collection of tuples. It is used to loop over the query result, one record at a time. The cursor is declared when the SQL query is **declared**. Later in the program, an **OPEN CURSOR** command fetches the query result from the database and sets the cursor to a position *before the first row* in the result of the query. This becomes the **current row** for the cursor. Subsequently, **FETCH** commands are issued in the program; each FETCH moves the cursor to the *next row* in the result of the query, making it the current row and copying its attribute values into the C (host language) program variables specified in the FETCH command by an INTO

clause. The cursor variable is basically an **iterator** that iterates (loops) over the tuples in the query result—one tuple at a time.

To determine when all the tuples in the result of the query have been processed, the communication variable SQLCODE (or, alternatively, SQLSTATE) is checked. If a FETCH command is issued that results in moving the cursor past the last tuple in the result of the query, a positive value (SQLCODE > 0) is returned in SQLCODE, indicating that no data (tuple) was found (or the string '02000' is returned in SQLSTATE). The programmer uses this to terminate the loop over the tuples in the query result. In general, numerous cursors can be opened at the same time. A **CLOSE CURSOR** command is issued to indicate that we are done with processing the result of the query associated with that cursor.

An example of using cursors to process a query result with multiple records is shown in Figure 10.3, where a cursor called EMP is declared in line 4. The EMP cursor is associated with the SQL query declared in lines 5 through 6, but the query is not executed until the OPEN EMP command (line 8) is processed. The OPEN <cursor name> command executes the query and fetches its result as a table into the program workspace, where the program can loop through the individual rows (tuples) by subsequent FETCH <cursor name> commands (line 9). We assume

---

Figure 10.3
Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.

```
    //Program Segment E2:
 0) prompt("Enter the Department Name: ", dname) ;
 1) EXEC SQL
 2)    SELECT Dnumber INTO :dnumber
 3)    FROM DEPARTMENT WHERE Dname = :dname ;
 4) EXEC SQL DECLARE EMP CURSOR FOR
 5)    SELECT Ssn, Fname, Minit, Lname, Salary
 6)    FROM EMPLOYEE WHERE Dno = :dnumber
 7)    FOR UPDATE OF Salary ;
 8) EXEC SQL OPEN EMP ;
 9) EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE = = 0) {
11)    printf("Employee name is:", Fname, Minit, Lname) ;
12)    prompt("Enter the raise amount: ", raise) ;
13)    EXEC SQL
14)       UPDATE EMPLOYEE
15)       SET Salary = Salary + :raise
16)       WHERE CURRENT OF EMP ;
17)    EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
18)    }
19) EXEC SQL CLOSE EMP ;
```

that appropriate C program variables have been declared as in Figure 10.1. The program segment in E2 reads (inputs) a department name (line 0), retrieves the matching department number from the database (lines 1 to 3), and then retrieves the employees who work in that department via the declared EMP cursor. A loop (lines 10 to 18) iterates over each record in the query result, one at a time, and prints the employee name, then reads (inputs) a raise amount for that employee (line 12) and updates the employee's salary in the database by the raise amount (lines 14 to 16).

This example also illustrates how the programmer can *update* database records. When a cursor is defined for rows that are to be modified (**updated**), we must add the clause **FOR UPDATE OF** in the cursor declaration and list the names of any attributes that will be updated by the program. This is illustrated in line 7 of code segment E2. If rows are to be **deleted**, the keywords **FOR UPDATE** must be added without specifying any attributes. In the embedded UPDATE (or DELETE) command, the condition **WHERE CURRENT OF** <cursor name> specifies that the current tuple referenced by the cursor is the one to be updated (or deleted), as in line 16 of E2.

There is no need to include the **FOR UPDATE OF** clause in line 7 of E2 if the results of the query are to be used *for retrieval purposes only* (no update or delete).

General Options for a Cursor Declaration. Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

> DECLARE <cursor name> [**INSENSITIVE**] [**SCROLL**] **CURSOR**
> [**WITH HOLD**] **FOR** <query specification>
> [**ORDER BY** <ordering specification>]
> [**FOR READ ONLY** | **FOR UPDATE** [**OF** <attribute list>]] ;

We already briefly discussed the options listed in the last line. The default is that the query is for retrieval purposes (FOR READ ONLY). If some of the tuples in the query result are to be updated, we need to specify FOR UPDATE OF <attribute list> and list the attributes that may be updated. If some tuples are to be deleted, we need to specify FOR UPDATE without any attributes listed.

When the optional keyword SCROLL is specified in a cursor declaration, it is possible to position the cursor in other ways than for purely sequential access. A **fetch orientation** can be added to the FETCH command, whose value can be one of NEXT, PRIOR, FIRST, LAST, ABSOLUTE *i*, and RELATIVE *i*. In the latter two commands, *i* must evaluate to an integer value that specifies an absolute tuple position within the query result (for ABSOLUTE *i*), or a tuple position relative to the current cursor position (for RELATIVE *i*). The default fetch orientation, which we used in our examples, is NEXT. The fetch orientation allows the programmer to move the cursor around the tuples in the query result with greater flexibility, providing random access by position or access in reverse order. When SCROLL is specified on the cursor, the general form of a FETCH command is as follows, with the parts in square brackets being optional:

> FETCH [[<fetch orientation>] **FROM**] <cursor name> **INTO** <fetch target list>;

The ORDER BY clause orders the tuples so that the FETCH command will fetch them in the specified order. It is specified in a similar manner to the corresponding clause for SQL queries (see Section 6.3.6). The last two options when declaring a cursor (INSENSITIVE and WITH HOLD) refer to transaction characteristics of database programs, which we will discuss in Chapter 20.

10.2.3 Specifying Queries at Runtime Using Dynamic SQL

In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, anytime we want to write a different query, we must modify the program code and go through all the steps involved (compiling, debugging, testing, and so on). In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*. For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs. Another example is when a user-friendly interface generates SQL queries dynamically for the user based on user input through a Web interface or mobile App. In this section, we give a brief overview of **dynamic SQL**, which is one technique for writing this type of database program, by giving a simple example to illustrate how dynamic SQL can work. In Section 10.3, we will describe another approach for dealing with dynamic queries using function libraries or class libraries.

Program segment E3 in Figure 10.4 reads a string that is input by the user (that string should be an SQL *update command* in this example) into the string program variable *sqlupdatestring* in line 3. It then **prepares** this as an SQL command in line 4 by associating it with the SQL variable *sqlcommand*. Line 5 then **executes** the command. Notice that in this case no syntax check or other types of checks on the command are possible *at compile time*, since the SQL command is not available until runtime. This contrasts with our previous examples of embedded SQL, where the query could be checked at compile time because its text was in the program source code.

In E3, the reason for separating PREPARE and EXECUTE is that if the command is to be executed multiple times in a program, it can be prepared only once. **Preparing the command** generally involves syntax and other types of checks by the system, as

```
   //Program Segment E3:
0) EXEC SQL BEGIN DECLARE SECTION ;
1) varchar sqlupdatestring [256] ;
2) EXEC SQL END DECLARE SECTION ;
   ...
3) prompt("Enter the Update Command: ", sqlupdatestring) ;
4) EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5) EXEC SQL EXECUTE sqlcommand ;
   ...
```

Figure 10.4
Program segment E3, a C program segment that uses dynamic SQL for updating a table.

well as generating the code for executing it. It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5 in E3) into a single statement by writing

```
EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring ;
```

This is useful if the command is to be executed only once. Alternatively, the programmer can separate the two statements to catch any errors after the PREPARE statement as in E3.

Although including a dynamic *update command* is relatively straightforward in dynamic SQL, a dynamic *retrieval query* is much more complicated. This is because the programmer does not know the types or the number of attributes to be retrieved by the SQL query when writing the program. A complex data structure is needed to allow for different numbers and types of attributes in the query result if no prior information is known about the dynamic query. Techniques similar to those that we shall discuss in Section 10.3 can be used to assign retrieval query results (and query parameters) to host program variables.

10.2.4 SQLJ: Embedding SQL Commands in Java

In the previous subsections, we gave an overview of how SQL commands can be embedded in a traditional programming language, using the C language in our examples. We now turn our attention to how SQL can be embedded in an object-oriented programming language,[8] in particular, the Java language. SQLJ is a standard that has been adopted by several vendors for embedding SQL in Java. Historically, SQLJ was developed after JDBC, which is used for accessing SQL databases from Java using class libraries and function calls. We discuss JDBC in Section 10.3.2. In this section, we focus on SQLJ as it is used in the Oracle RDBMS. An SQLJ translator will generally convert SQL statements into Java, which can then be executed through the JDBC interface. Hence, it is necessary to install a *JDBC driver* when using SQLJ.[9] In this section, we focus on how to use SQLJ concepts to write embedded SQL in a Java program.

Before being able to process SQLJ with Java in Oracle, it is necessary to import several class libraries, shown in Figure 10.5. These include the JDBC and IO classes (lines 1 and 2), plus the additional classes listed in lines 3, 4, and 5. In addition, the program must first connect to the desired database using the function call `getConnection`, which is one of the methods of the `oracle` class in line 5 of Figure 10.5. The format of this function call, which returns an object of type *default context*,[10] is as follows:

```
public static DefaultContext
getConnection(String url, String user, String password,
    Boolean autoCommit)
throws SQLException ;
```

---

[8]This section assumes familiarity with object-oriented concepts (see Chapter 12) and basic Java concepts.

[9]We discuss JDBC drivers in Section 10.3.2.

[10]A *default context*, when set, applies to subsequent commands in the program until it is changed.

```
1)  import java.sql.* ;
2)  import java.io.* ;
3)  import sqlj.runtime.* ;
4)  import sqlj.runtime.ref.* ;
5)  import oracle.sqlj.runtime.* ;
    ...
6)  DefaultContext cntxt =
7)  oracle.getConnection("<url name>", "<user name>", "<password>", true) ;
8)  DefaultContext.setDefaultContext(cntxt) ;
    ...
```

Figure 10.5
Importing classes needed for including SQLJ in Java programs in Oracle, and establishing a connection and default context.

For example, we can write the statements in lines 6 through 8 in Figure 10.5 to connect to an Oracle database located at the url <url name> using the login of <user name> and <password> with automatic commitment of each command,[11] and then set this connection as the **default context** for subsequent commands.

In the following examples, we will not show complete Java classes or programs since it is not our intention to teach Java. Rather, we will show program segments that illustrate the use of SQLJ. Figure 10.6 shows the Java program variables used in our examples. Program segment J1 in Figure 10.7 reads an employee's Ssn and prints some of the employee's information from the database.

Notice that because Java already uses the concept of **exceptions** for error handling, a special exception called SQLException is used to return errors or exception conditions after executing an SQL database command. This plays a similar role to SQLCODE and SQLSTATE in embedded SQL. Java has many types of predefined exceptions. Each Java operation (function) must specify the exceptions that can be **thrown**—that is, the exception conditions that may occur while executing the Java code of that operation. If a defined exception occurs, the system transfers control to the Java code specified for exception handling. In J1, exception handling for an SQLException is specified in lines 7 and 8. In Java, the following structure

```
try {<operation>} catch (<exception>) {<exception handling
    code>} <continuation code>
```

Figure 10.6
Java program variables used in SQLJ examples J1 and J2.

```
1)  string dname, ssn , fname, fn, lname, ln,
    bdate, address ;
2)  char sex, minit, mi ;
3)  double salary, sal ;
4)  integer dno, dnumber ;
```

[11]*Automatic commitment* roughly means that each command is applied to the database after it is executed. The alternative is that the programmer wants to execute several related database commands and then commit them together. We discuss commit concepts in Chapter 20 when we describe database transactions.

```
     //Program Segment J1:
  1) ssn = readEntry("Enter a Social Security Number: ") ;
  2) try {
  3)   #sql { SELECT Fname, Minit, Lname, Address, Salary
  4)     INTO :fname, :minit, :lname, :address, :salary
  5)     FROM EMPLOYEE WHERE Ssn = :ssn} ;
  6) } catch (SQLException se) {
  7)     System.out.println("Social Security Number does not exist: " + ssn) ;
  8)     Return ;
  9)   }
 10) System.out.println(fname + " " + minit + " " + lname + " " + address
        + " " + salary)
```

Figure 10.7
Program segment J1,
a Java program
segment with SQLJ.

is used to deal with exceptions that occur during the execution of <operation>. If no exception occurs, the <continuation code> is processed directly. Exceptions that can be thrown by the code in a particular operation should be specified as part of the operation declaration or *interface*—for example, in the following format:

```
<operation return type> <operation name> (<parameters>)
    throws SQLException, IOException ;
```

In SQLJ, the embedded SQL commands within a Java program are preceded by #sql, as illustrated in J1 line 3, so that they can be identified by the preprocessor. The #sql is used instead of the keywords EXEC SQL that are used in embedded SQL with the C programming language (see Section 10.2.1). SQLJ uses an *INTO clause*—similar to that used in embedded SQL—to return the attribute values retrieved from the database by an SQL query into Java program variables. The program variables are preceded by colons (:) in the SQL statement, as in embedded SQL.

In J1 a *single tuple* is retrieved by the embedded SQLJ query; that is why we are able to assign its attribute values directly to Java program variables in the INTO clause in line 4 in Figure 10.7. For queries that retrieve many tuples, SQLJ uses the concept of an *iterator*, which is similar to a cursor in embedded SQL.

10.2.5 Processing Query Results in SQLJ Using Iterators

In SQLJ, an **iterator** is a type of object associated with a collection (set or multiset) of records in a **query result**.[12] The iterator is associated with the tuples and attributes that appear in a query result. There are two types of iterators:

1. A **named iterator** is associated with a query result by listing the attribute *names and types* that appear in the query result. The attribute names must correspond to appropriately declared Java program variables, as shown in Figure 10.6.

2. A **positional iterator** lists only the *attribute types* that appear in the query result.

---

[12]We shall discuss iterators in more detail in Chapter 12 when we present object database concepts.

In both cases, the list should be *in the same order* as the attributes that are listed in the SELECT clause of the query. However, looping over a query result is different for the two types of iterators. First, we show an example of using a *named* iterator in Figure 10.8, program segment J2A. Line 9 in Figure 10.8 shows how a *named itera-tor type* Emp is declared. Notice that the names of the attributes in a named iterator type must match the names of the attributes in the SQL query result. Line 10 shows how an *iterator object* e of type Emp is created in the program and then associated with a query (lines 11 and 12).

When the iterator object is associated with a query (lines 11 and 12 in Figure 10.8), the program fetches the query result from the database and sets the iterator to a position *before the first row* in the result of the query. This becomes the **current row** for the iterator. Subsequently, **next** operations are issued on the iterator object; each next moves the iterator to the *next row* in the result of the query, making it the current row. If the row exists, the operation retrieves the attribute values for that row into the corresponding program variables. If no more rows exist, the next operation returns NULL, and can thus be used to control the looping. Notice that the named iterator *does not need* an INTO clause, because the program variables corre-sponding to the retrieved attributes are already specified when the iterator type is declared (line 9 in Figure 10.8).

In Figure 10.8, the command (e.next()) in line 13 performs two functions: It gets the next tuple in the query result and controls the WHILE loop. Once the

Figure 10.8
Program segment J2A, a Java program segment that uses a named iterator to print employee information in a particular department.

```
    //Program Segment J2A:
 0) dname = readEntry("Enter the Department Name: ") ;
 1) try {
 2)    #sql { SELECT Dnumber INTO :dnumber
 3)      FROM DEPARTMENT WHERE Dname = :dname} ;
 4) } catch (SQLException se) {
 5)    System.out.println("Department does not exist: " + dname) ;
 6)    Return ;
 7)    }
 8) System.out.printline("Employee information for Department: " + dname) ;
 9) #sql iterator Emp(String ssn, String fname, String minit, String lname,
       double salary) ;
10) Emp e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12)    FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) while (e.next()) {
14)    System.out.printline(e.ssn + " " + e.fname + " " + e.minit + " " " +
          e.lname  + " " + e.salary) ;
15) } ;
16) e.close() ;
```

program is done with processing the query result, the command e.close() (line 16) closes the iterator.

Next, consider the same example using *positional* iterators as shown in Figure 10.9 (program segment J2B). Line 9 in Figure 10.9 shows how a *positional iterator type* Emppos is declared. The main difference between this and the named iterator is that there are no attribute names (corresponding to program variable names) in the positional iterator—only attribute types. This can provide more flexibility, but it makes the processing of the query result slightly more complex. The attribute types must still be compatible with the attribute types in the SQL query result and in the same order. Line 10 shows how a *positional iterator object* e of type Emppos is created in the program and then associated with a query (lines 11 and 12).

The positional iterator behaves in a manner that is more similar to embedded SQL (see Section 10.2.2). A **FETCH <iterator variable> INTO <program variables>** command is needed to get the next tuple in a query result. The first time fetch is executed, it gets the first tuple (line 13 in Figure 10.9). Line 16 gets the next tuple until no more tuples exist in the query result. To control the loop, a positional iterator function e.endFetch() is used. This function is automatically set to a value of TRUE when the iterator is initially associated with an SQL query (line 11), and is set to FALSE each time a fetch command returns a valid tuple from the query result. It is set to TRUE again when a fetch command does not find any more tuples. Line 14 shows how the looping is controlled by negation.

Figure 10.9
Program segment J2B, a Java program segment that uses a positional iterator to print employee information in a particular department.

```
    //Program Segment J2B:
 0) dname = readEntry("Enter the Department Name: ") ;
 1) try {
 2)    #sql { SELECT Dnumber INTO :dnumber
 3)       FROM DEPARTMENT WHERE Dname = :dname} ;
 4) } catch (SQLException se) {
 5)    System.out.println("Department does not exist: " + dname) ;
 6)    Return ;
 7)    }
 8) System.out.printline("Employee information for Department: " + dname) ;
 9) #sql iterator Emppos(String, String, String, String, double) ;
10) Emppos e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12)    FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)    System.out.printline(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)    #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```

10.3 Database Programming with Function Calls and Class Libraries: SQL/CLI and JDBC

Embedded SQL (see Section 10.2) is sometimes referred to as a **static** database programming approach because the query text is written within the program source code and cannot be changed without recompiling or reprocessing the source code. The use of function calls is a more **dynamic** approach for database programming than embedded SQL. We already saw one dynamic database programming technique—dynamic SQL—in Section 10.2.3. The techniques discussed here provide another approach to dynamic database programming. A **library of functions**, also known as an **application programming interface** (**API**), is used to access the database. Although this provides more flexibility because no preprocessor is needed, one drawback is that syntax and other checks on SQL commands have to be done at runtime. Another drawback is that it sometimes requires more complex programming to access query results because the types and numbers of attributes in a query result may not be known in advance.

In this section, we give an overview of two function call interfaces. We first discuss the **SQL Call Level Interface (SQL/CLI)**, which is part of the SQL standard. This was developed as a standardization of the popular library of functions known as **ODBC (Open Database Connectivity)**. We use C as the host language in our SQL/CLI examples. Then we give an overview of **JDBC**, which is the call function interface for accessing databases from Java. Although it is commonly assumed that JDBC stands for Java Database Connectivity, JDBC is just a registered trademark of Sun Microsystems (now Oracle), *not* an acronym.

The main advantage of using a function call interface is that it makes it easier to access multiple databases within the same application program, even if they are stored under different DBMS packages. We discuss this further in Section 10.3.2 when we discuss Java database programming with JDBC, although this advantage also applies to database programming with SQL/CLI and ODBC (see Section 10.3.1).

10.3.1 Database Programming with SQL/CLI Using C as the Host Language

Before using the function calls in SQL/CLI, it is necessary to install the appropriate library packages on the database server. These packages are obtained from the vendor of the DBMS being used. We now give an overview of how SQL/CLI can be used in a C program.[13] We will illustrate our presentation with the sample program segment CLI1 shown in Figure 10.10.

---

[13]Our discussion here also applies to the C++ and C# programming languages, since we do not use any of the object-oriented features but focus on the database programming mechanism.

```
    //Program CLI1:
0) #include sqlcli.h ;
1) void printSal() {
2) SQLHSTMT stmt1 ;
3) SQLHDBC con1 ;
4) SQLHENV env1 ;
5) SQLRETURN ret1, ret2, ret3, ret4 ;
6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
      SQL_NTS) else exit ;
9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Ssn = ?",
      SQL_NTS) ;
11) prompt("Enter a Social Security Number: ", ssn) ;
12) SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   if (!ret2) printf(ssn, lname, salary)
19)     else printf("Social Security Number does not exist: ", ssn) ;
20)   }
21) }
```

Figure 10.10
Program segment CLI1, a C program segment with SQL/CLI.

Handles to environment, connection, statement, and description records. When using SQL/CLI, the SQL statements are dynamically created and passed as *string parameters* in the function calls. Hence, it is necessary to keep track of the information about host program interactions with the database in runtime data structures because the database commands are processed at runtime. The information is kept in four types of records, represented as *structs* in C data types. An **environment record** is used as a container to keep track of one or more database connections and to set environment information. A **connection record** keeps track of the information needed for a particular database connection. A **statement record** keeps track of the information needed for one SQL statement. A **description record** keeps track of the information about tuples or parameters—for example, the number of attributes and their types in a tuple, or the number and types of parameters in a function call. This is needed when the programmer does not know this information about the query when writing the program. In our examples, we assume that the programmer knows the exact query, so we do not show any description records.

Each record is accessible to the program through a C pointer variable—called a **handle** to the record. The handle is returned when a record is first created. To create a record and return its handle, the following SQL/CLI function is used:

```
SQLAllocHandle(<handle_type>, <handle_1>, <handle_2>)
```

In this function, the parameters are as follows:

- `<handle_type>` indicates the type of record being created. The possible values for this parameter are the keywords `SQL_HANDLE_ENV`, `SQL_HANDLE_DBC`, `SQL_HANDLE_STMT`, or `SQL_HANDLE_DESC`, for an environment, connection, statement, or description record, respectively.

- `<handle_1>` indicates the container within which the new handle is being created. For example, for a connection record this would be the environment within which the connection is being created, and for a statement record this would be the connection for that statement.

- `<handle_2>` is the pointer (handle) to the newly created record of type `<handle_type>`.

Steps in a database program. When writing a C program that will include database calls through SQL/CLI, the following are the typical steps that are taken. We illustrate the steps by referring to the example CLI1 in Figure 10.10, which reads a Social Security number of an employee and prints the employee's last name and salary.

1. **Including the library of functions.** The *library of functions* comprising SQL/CLI must be included in the C program. This is called sqlcli.h, and is included using line 0 in Figure 10.10.

2. **Declaring handle variables.** Declare *handle variables* of types `SQLHSTMT`, `SQLHDBC`, `SQLHENV`, and `SQLHDESC` for the statements, connections, environments, and descriptions needed in the program, respectively (lines 2 to 4).[14] Also declare variables of type `SQLRETURN` (line 5) to hold the return codes from the SQL/CLI function calls. A return code of 0 (zero) indicates *successful execution* of the function call.

3. **Environment record.** An *environment record* must be set up in the program using `SQLAllocHandle`. The function to do this is shown in line 6. Because an environment record is not contained in any other record, the parameter `<handle_1>` is the NULL handle `SQL_NULL_HANDLE` (NULL pointer) when creating an environment. The handle (pointer) to the newly created environment record is returned in variable `env1` in line 6.

4. **Connecting to the database.** A *connection record* is set up in the program using `SQLAllocHandle`. In line 7, the connection record created has the handle `con1` and is contained in the environment `env1`. A **connection** is then established in `con1` to a particular server database using the `SQLConnect`

---

[14]To keep our presentation simple, we will not show description records here.

function of SQL/CLI (line 8). In our example, the database server name we are connecting to is *dbs* and the account name and password for login are *js* and *xyz*, respectively.

5. **Statement record.** A *statement record* is set up in the program using `SQLAllocHandle`. In line 9, the statement record created has the handle `stmt1` and uses the connection `con1`.

6. **Preparing an SQL statement and statement parameters.** The SQL statement is *prepared* using the SQL/CLI function **`SQLPrepare`**. In line 10, this assigns the SQL **statement string** (the *query* in our example) to the statement handle `stmt1`. The question mark (`?`) symbol in line 10 represents a **statement parameter**, which is a value to be determined at runtime—typically by binding it to a C program variable. In general, there could be several parameters in a statement string. They are distinguished by the order of appearance of the question marks in the statement string (the first `?` represents parameter 1, the second `?` represents parameter 2, and so on). The last parameter in `SQLPrepare` should give the length of the SQL statement string in bytes, but if we enter the keyword `SQL_NTS`, this indicates that the string holding the query is a *NULL-terminated string* so that SQL can calculate the string length automatically. This use of `SQL_NTS` also applies to *other string parameters* in the function calls in our examples.

7. **Binding the statement parameters.** Before executing the query, any parameters in the query string should be bound to program variables using the SQL/CLI function `SQLBindParameter`. In Figure 10.10, the parameter (indicated by `?`) to the prepared query referenced by `stmt1` is bound to the C program variable `ssn` in line 12. If there are *n* parameters in the SQL statement, we should have *n* `SQLBindParameter` function calls, each with a different *parameter position* (1, 2, ... , *n*).

8. **Executing the statement.** Following these preparations, we can now execute the SQL statement referenced by the handle `stmt1` using the function `SQLExecute` (line 13). Notice that although the query will be executed in line 13, the query results have not yet been assigned to any C program variables.

9. **Processing the query result.** In order to determine where the result of the query is returned, one common technique is the **bound columns** approach. Here, each column in a query result is bound to a C program variable using the `SQLBindCol` function. The columns are distinguished by their order of appearance in the SQL query. In Figure 10.10 lines 15 and 16, the two columns in the query (`Lname` and `Salary`) are bound to the C program variables `lname` and `salary`, respectively.[15]

---

[15]An alternative technique known as **unbound columns** uses different SQL/CLI functions, namely SQLGetCol or SQLGetData, to retrieve columns from the query result without previously binding them; these are applied after the SQLFetch command in line 17.

10. **Retrieving column values.** Finally, in order to retrieve the column values into the C program variables, the function SQLFetch is used (line 17). This function is similar to the FETCH command of embedded SQL. If a query result has a collection of tuples, each SQLFetch call gets the next tuple and returns its column values into the bound program variables. SQLFetch returns an exception (nonzero) code if there are no more tuples in the query result.[16]

As we can see, using dynamic function calls requires a lot of preparation to set up the SQL statements and to bind statement parameters and query results to the appropriate program variables.

In CLI1 a *single tuple* is selected by the SQL query. Figure 10.11 shows an example of retrieving multiple tuples. We assume that appropriate C program variables have been declared as in Figure 10.1. The program segment in CLI2 reads (inputs) a

```
//Program Segment CLI2:
0) #include sqlcli.h ;
1) void printDepartmentEmps() {
2) SQLHSTMT stmt1 ;
3) SQLHDBC con1 ;
4) SQLHENV env1 ;
5) SQLRETURN ret1, ret2, ret3, ret4 ;
6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
     SQL_NTS) else exit ;
9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Dno = ?",
     SQL_NTS) ;
11) prompt("Enter the Department Number: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)    SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)    SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)    ret2 = SQLFetch(stmt1) ;
18)    while (!ret2) {
19)      printf(lname, salary) ;
20)      ret2 = SQLFetch(stmt1) ;
21)      }
22)    }
23) }
```

Figure 10.11
Program segment CLI2, a C program segment that uses SQL/CLI for a query with a collection of tuples in its result.

---

[16]If unbound program variables are used, SQLFetch returns the tuple into a temporary program area. Each subsequent SQLGetCol (or SQLGetData) returns one attribute value in order. Basically, for each row in the query result, the program should iterate over the attribute values (columns) in that row. This is useful if the number of columns in the query result is variable.

department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary.

10.3.2 JDBC: SQL Class Library for Java Programming

We now turn our attention to how SQL can be called from the Java object-oriented programming language.[17] The class libraries and associated function calls for this access are known as **JDBC**.[18] The Java programming language was designed to be platform independent—that is, a program should be able to run on any type of computer system that has a Java interpreter installed. Because of this portability, many RDBMS vendors provide JDBC drivers so that it is possible to access their systems via Java programs.

JDBC drivers. A **JDBC driver** is basically an implementation of the classes and associated objects and function calls specified in JDBC for a particular vendor's RDBMS. Hence, a Java program with JDBC objects and function calls can access any RDBMS that has a JDBC driver available.

Because Java is object-oriented, its function libraries are implemented as **classes**. Before being able to process JDBC function calls with Java, it is necessary to import the **JDBC class libraries**, which are called `java.sql.*`. These can be downloaded and installed via the Web.[19]

JDBC is designed to allow a single Java program to connect to several different databases. These are sometimes called the **data sources** accessed by the Java program, and could be stored using RDBMSs from different vendors residing on different machines. Hence, different data source accesses within the same Java program may require JDBC drivers from different vendors. To achieve this flexibility, a special JDBC class called the **driver manager** class is employed, which keeps track of the installed drivers. A driver should be *registered* with the driver manager before it is used. The operations (methods) of the driver manager class include `getDriver`, `registerDriver`, and `deregisterDriver`. These can be used to add and remove drivers for different systems dynamically. Other functions set up and close connections to data sources.

To load a JDBC driver explicitly, the generic Java function for loading a class can be used. For example, to load the JDBC driver for the Oracle RDBMS, the following command can be used:

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

---

[17]This section assumes familiarity with object-oriented concepts (see Chapter 11) and basic Java concepts.

[18]As we mentioned earlier, JDBC is a registered trademark of Sun Microsystems, although it is commonly thought to be an acronym for Java Database Connectivity.

[19]These are available from several Web sites—for example, at http://industry.java.sun.com/products/jdbc/drivers.

This will register the driver with the driver manager and make it available to the program. It is also possible to load and register the driver(s) needed in the command line that runs the program, for example, by including the following in the command line:

```
-Djdbc.drivers = oracle.jdbc.driver
```

JDBC programming steps. The following are typical steps that are taken when writing a Java application program with database access through JDBC function calls. We illustrate the steps by referring to the example JDBC1 in Figure 10.12, which reads a Social Security number of an employee and prints the employee's last name and salary.

1. **Import the JDBC class library.** The JDBC *library of classes* must be imported into the Java program. These classes are called `java.sql.*`, and can be imported using line 1 in Figure 10.12. Any additional Java class libraries needed by the program must also be imported.

Figure 10.12
Program segment JDBC1, a Java program segment with JDBC.

```
   //Program JDBC1:
0) import java.io.* ;
1) import java.sql.*
   ...
2) class getEmpInfo {
3)   public static void main (String args []) throws SQLException, IOException {
4)     try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)     } catch (ClassNotFoundException x) {
6)       System.out.println ("Driver could not be loaded") ;
7)     }
8)     String dbacct, passwrd, ssn, lname ;
9)     Double salary ;
10)    dbacct = readentry("Enter database account:") ;
11)    passwrd = readentry("Enter password:") ;
12)    Connection conn = DriverManager.getConnection
13)      ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
14)    String stmt1 = "select Lname, Salary from EMPLOYEE where Ssn = ?" ;
15)    PreparedStatement p = conn.prepareStatement(stmt1) ;
16)    ssn = readentry("Enter a Social Security Number: ") ;
17)    p.clearParameters() ;
18)    p.setString(1, ssn) ;
19)    ResultSet r = p.executeQuery() ;
20)    while (r.next()) {
21)      lname = r.getString(1) ;
22)      salary = r.getDouble(2) ;
23)      system.out.printline(lname + salary) ;
24)   } }
25) }
```

2. **Load the JDBC driver.** This is shown in lines 4 to 7. The Java exception in line 5 occurs if the driver is not loaded successfully.

3. **Create appropriate variables.** These are the variables needed in the Java program (lines 8 and 9).

4. **The `Connection` object.** A **connection object** is created using the `getConnection` function of the `DriverManager` class of JDBC. In lines 12 and 13, the `Connection` object is created by using the function call `getConnection(urlstring)`, where `urlstring` has the form

   ```
   jdbc:oracle:<driverType>:<dbaccount>/<password>
   ```

   An alternative form is

   ```
   getConnection(url, dbaccount, password)
   ```

   Various properties can be set for a connection object, but they are mainly related to transactional properties, which we discuss in Chapter 21.

5. **The Prepared `Statement` object.** A **statement object** is created in the program. In JDBC, there is a basic statement class, `Statement`, with two specialized subclasses: `PreparedStatement` and `CallableStatement`. The example in Figure 10.12 illustrates how **`PreparedStatement`** objects are created and used. The next example (Figure 10.13) illustrates the other type of `Statement` objects. In line 14 in Figure 10.12, a query string with a single parameter—indicated by the `?` symbol—is created in the string variable `stmt1`. In line 15, an object `p` of type `PreparedStatement` is created based on the query string in `stmt1` and using the connection object `conn`. In general, the programmer should use `PreparedStatement` objects if a query is to be executed *multiple times*, since it would be prepared, checked, and compiled only once, thus saving this cost for the additional executions of the query.

6. **Setting the statement parameters.** The question mark (`?`) symbol in line 14 represents a **statement parameter**, which is a value to be determined at runtime, typically by binding it to a Java program variable. In general, there could be several parameters, distinguished by the order of appearance of the question marks within the statement string (first `?` represents parameter 1, second `?` represents parameter 2, and so on), as we discussed previously.

7. **Binding the statement parameters.** Before executing a `PreparedStatement` query, any parameters should be bound to program variables. Depending on the type of the parameter, different functions such as `setString`, `setInteger`, `setDouble`, and so on are applied to the `PreparedStatement` object to set its parameters. The appropriate function should be used to correspond to the data type of the parameter being set. In Figure 10.12, the parameter (indicated by `?`) in object `p` is bound to the Java program variable `ssn` in line 18. The function `setString` is used because `ssn` is a string variable. If there are *n* parameters in the SQL statement, we should have *n* `set ...` functions, each with a different parameter position (1, 2, ... , *n*). Generally, it is advisable to clear all parameters before setting any new values (line 17).

```
      //Program Segment JDBC2:
 0) import java.io.* ;
 1) import java.sql.*
     ...
 2) class printDepartmentEmps {
 3)   public static void main (String args [])
          throws SQLException, IOException {
 4)      try { Class.forName("oracle.jdbc.driver.OracleDriver")
 5)      } catch (ClassNotFoundException x) {
 6)        System.out.println ("Driver could not be loaded") ;
 7)      }
 8)      String dbacct, passwrd, lname ;
 9)      Double salary ;
10)      Integer dno ;
11)      dbacct = readentry("Enter database account:") ;
12)      passwrd = readentry("Enter password:") ;
13)      Connection conn = DriverManager.getConnection
14)        ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
15)      dno = readentry("Enter a Department Number: ") ;
16)      String q = "select Lname, Salary from EMPLOYEE where Dno = " +
          dno.tostring() ;
17)      Statement s = conn.createStatement() ;
18)      ResultSet r = s.executeQuery(q) ;
19)      while (r.next()) {
20)        lname = r.getString(1) ;
21)        salary = r.getDouble(2) ;
22)        system.out.printline(lname + salary) ;
23)   } }
24) }
```

Figure 10.13
Program segment JDBC2, a Java program segment that uses JDBC for a query with a collection of tuples in its result.

8. **Executing the SQL statement.** Following these preparations, we can now execute the SQL statement referenced by the object p using the function executeQuery (line 19). There is a generic function execute in JDBC, plus two specialized functions: executeUpdate and executeQuery. executeUpdate is used for SQL insert, delete, or update statements, and returns an integer value indicating the number of tuples that were affected. executeQuery is used for SQL retrieval statements, and returns an object of type ResultSet, which we discuss next.

9. **Processing the ResultSet object.** In line 19, the result of the query is returned in an *object r* of type **ResultSet**. This resembles a two-dimensional array or a table, where the tuples are the rows and the attributes returned are the columns. A ResultSet object is similar to a cursor in embedded SQL and an iterator in SQLJ. In our example, when the query is executed, r refers to a tuple before the first tuple in the query result. The r.next() function (line 20) moves to the next tuple (row) in the ResultSet object and returns NULL if there are no more objects. This is used to control the looping. The

programmer can refer to the attributes in the current tuple using various `get ...` functions that depend on the type of each attribute (for example, `getString`, `getInteger`, `getDouble`, and so on). The programmer can either use the attribute positions (1, 2) or the actual attribute names (`"Lname"`, `"Salary"`) with the `get` ... functions. In our examples, we used the positional notation in lines 21 and 22.

In general, the programmer can check for SQL exceptions after each JDBC function call. We did not do this to simplify the examples.

Notice that JDBC does not distinguish between queries that return single tuples and those that return multiple tuples, unlike some of the other techniques. This is justifiable because a single tuple result set is just a special case.

In example JDBC1, a *single tuple* is selected by the SQL query, so the loop in lines 20 to 24 is executed at most once. The example shown in Figure 10.13 illustrates the retrieval of multiple tuples. The program segment in JDBC2 reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary. This example also illustrates how we can *execute a query directly*, without having to prepare it as in the previous example. This technique is preferred for queries that will be executed only once, since it is simpler to program. In line 17 of Figure 10.13, the programmer creates a **Statement** object (instead of `PreparedStatement`, as in the previous example) without associating it with a particular query string. The query string q is *passed to the statement object* `s` when it is executed in line 18.

This concludes our brief introduction to JDBC. The interested reader is referred to the Web site http://java.sun.com/docs/books/tutorial/jdbc/, which contains many further details about JDBC.

10.4 Database Stored Procedures and SQL/PSM

This section introduces two additional topics related to database programming. In Section 10.4.1, we discuss the concept of stored procedures, which are program modules that are stored by the DBMS at the database server. Then in Section 10.4.2 we discuss the extensions to SQL that are specified in the standard to include general-purpose programming constructs in SQL. These extensions are known as SQL/PSM (SQL/Persistent Stored Modules) and can be used to write stored procedures. SQL/PSM also serves as an example of a database programming language that extends a database model and language—namely, SQL—with programming language constructs, such as conditional statements and loops.

10.4.1 Database Stored Procedures and Functions

In our presentation of database programming techniques so far, there was an implicit assumption that the database application program was running on a client

machine, or more likely at the *application server computer* in the middle-tier of a three-tier client-server architecture (see Section 2.5.4 and Figure 2.7). In either case, the machine where the program is executing is different from the machine on which the database server—and the main part of the DBMS software package—is located. Although this is suitable for many applications, it is sometimes useful to create database program modules—procedures or functions—that are stored and executed by the DBMS at the database server. These are historically known as database **stored procedures**, although they can be functions or procedures. The term used in the SQL standard for stored procedures is **persistent stored modules** because these programs are stored persistently by the DBMS, similarly to the persistent data stored by the DBMS.

Stored procedures are useful in the following circumstances:

- If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.

- Executing a program at the server can reduce data transfer and communication cost between the client and server in certain situations.

- These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users via the stored procedures. Additionally, they can be used to check for complex constraints that are beyond the specification power of assertions and triggers.

In general, many commercial DBMSs allow stored procedures and functions to be written in a general-purpose programming language. Alternatively, a stored procedure can be made of simple SQL commands such as retrievals and updates. The general form of declaring stored procedures is as follows:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;
```

The parameters and local declarations are optional, and are specified only if needed. For declaring a function, a return type is necessary, so the declaration form is:

```
CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body> ;
```

If the procedure (or function) is written in a general-purpose programming language, it is typical to specify the language as well as a file name where the program code is stored. For example, the following format can be used:

```
CREATE PROCEDURE <procedure name> (<parameters>)
LANGUAGE <programming language name>
EXTERNAL NAME <file path name> ;
```

In general, each parameter should have a **parameter type** that is one of the SQL data types. Each parameter should also have a **parameter mode**, which is one of IN, OUT, or INOUT. These correspond to parameters whose values are input only, output (returned) only, or both input and output, respectively.

Because the procedures and functions are stored persistently by the DBMS, it should be possible to call them from the various SQL interfaces and programming techniques. The CALL **statement** in the SQL standard can be used to invoke a stored procedure—either from an interactive interface or from embedded SQL or SQLJ. The format of the statement is as follows:

CALL <procedure or function name> (<argument list>) ;

If this statement is called from JDBC, it should be assigned to a statement object of type **CallableStatement** (see Section 10.3.2).

10.4.2 SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

SQL/PSM is the part of the SQL standard that specifies how to write persistent stored modules. It includes the statements to create functions and procedures that we described in the previous section. It also includes additional programming constructs to enhance the power of SQL for the purpose of writing the code (or body) of stored procedures and functions.

In this section, we discuss the SQL/PSM constructs for conditional (branching) statements and for looping statements. These will give a flavor of the type of constructs that SQL/PSM has incorporated;[20] then we give an example to illustrate how these constructs can be used.

The conditional branching statement in SQL/PSM has the following form:

IF <condition> THEN <statement list>
    ELSEIF <condition> THEN <statement list>
    …
    ELSEIF <condition> THEN <statement list>
    ELSE <statement list>
END IF ;

Consider the example in Figure 10.14, which illustrates how the conditional branch structure can be used in an SQL/PSM function. The function returns a string value (line 1) describing the size of a department within a company based on the number of employees. There is one IN integer parameter, deptno, which gives a department number. A local variable NoOfEmps is declared in line 2. The query in lines 3 and 4 returns the number of employees in the department, and the conditional

---

[20]We only give a brief introduction to SQL/PSM here. There are many other features in the SQL/PSM standard.

```
//Function PSM1:
0) CREATE FUNCTION Dept_size(IN deptno INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE No_of_emps INTEGER ;
3) SELECT COUNT(*) INTO No_of_emps
4) FROM EMPLOYEE WHERE Dno = deptno ;
5) IF No_of_emps > 100 THEN RETURN "HUGE"
6) ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7) ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8) ELSE RETURN "SMALL"
9) END IF ;
```

Figure 10.14
Declaring a function in
SQL/PSM.

branch in lines 5 to 8 then returns one of the values {'HUGE', 'LARGE', 'MEDIUM', 'SMALL'} based on the number of employees.

SQL/PSM has several constructs for looping. There are standard while and repeat looping structures, which have the following forms:

```
WHILE <condition> DO
      <statement list>
END WHILE ;
REPEAT
      <statement list>
UNTIL <condition>
END REPEAT ;
```

There is also a cursor-based looping structure. The statement list in such a loop is executed once for each tuple in the query result. This has the following form:

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
      <statement list>
END FOR ;
```

Loops can have names, and there is a LEAVE <loop name> statement to break a loop when a condition is satisfied. SQL/PSM has many other features, but they are outside the scope of our presentation.

10.5 Comparing the Three Approaches

In this section, we briefly compare the three approaches for database programming and discuss the advantages and disadvantages of each approach.

4. **Embedded SQL Approach.** The main advantage of this approach is that the query text is part of the program source code itself, and hence can be checked for syntax errors and validated against the database schema at compile time. This also makes the program quite readable, as the queries are readily visible

in the source code. The main disadvantages are the loss of flexibility in changing the query at runtime, and the fact that all changes to queries must go through the whole recompilation process. In addition, because the queries are known beforehand, the choice of program variables to hold the query results is a simple task, and so the programming of the application is generally easier. However, for complex applications where queries have to be generated at runtime, the function call approach will be more suitable.

5. **Library of Classes and Function Calls Approach.** This approach provides more flexibility in that queries can be generated at runtime if needed. However, this leads to more complex programming, as program variables that match the columns in the query result may not be known in advance. Because queries are passed as statement strings within the function calls, no checking can be done at compile time. All syntax checking and query validation has to be done at runtime by preparing the query, and the programmer must check and account for possible additional runtime errors within the program code.

6. **Database Programming Language Approach.** This approach does not suffer from the impedance mismatch problem, as the programming language data types are the same as the database data types. However, programmers must learn a new programming language rather than use a language they are already familiar with. In addition, some database programming languages are vendor-specific, whereas general-purpose programming languages can easily work with systems from multiple vendors.

10.6 Summary

In this chapter we presented additional features of the SQL database language. In particular, we presented an overview of the most important techniques for database programming in Section 10.1. Then we discussed the various approaches to database application programming in Sections 10.2 to 10.4.

In Section 10.2, we discussed the general technique known as embedded SQL, where the queries are part of the program source code. A precompiler is typically used to extract SQL commands from the program for processing by the DBMS, and replacing them with function calls to the DBMS compiled code. We presented an overview of embedded SQL, using the C programming language as host language in our examples. We also discussed the SQLJ technique for embedding SQL in Java programs. The concepts of cursor (for embedded SQL) and iterator (for SQLJ) were presented and illustrated by examples to show how they are used for looping over the tuples in a query result, and extracting the attribute value into program variables for further processing.

In Section 10.3, we discussed how function call libraries can be used to access SQL databases. This technique is more dynamic than embedding SQL, but requires more complex programming because the attribute types and number in a query result may be determined at runtime. An overview of the SQL/CLI standard was

presented, with examples using C as the host language. We discussed some of the functions in the SQL/CLI library, how queries are passed as strings, how query parameters are assigned at runtime, and how results are returned to program variables. We then gave an overview of the JDBC class library, which is used with Java, and discussed some of its classes and operations. In particular, the `ResultSet` class is used to create objects that hold the query results, which can then be iterated over by the `next()` operation. The get and set functions for retrieving attribute values and setting parameter values were also discussed.

In Section 10.4, we gave a brief overview of stored procedures, and discussed SQL/PSM as an example of a database programming language. Finally, we briefly compared the three approaches in Section 10.5. It is important to note that we chose to give a comparative overview of the three main approaches to database programming, since studying a particular approach in depth is a topic that is worthy of its own textbook.

Review Questions

10.1. What is ODBC? How is it related to SQL/CLI?

10.2. What is JDBC? Is it an example of embedded SQL or of using function calls?

10.3. List the three main approaches to database programming. What are the advantages and disadvantages of each approach?

10.4. What is the impedance mismatch problem? Which of the three programming approaches minimizes this problem?

10.5. Describe the concept of a cursor and how it is used in embedded SQL.

10.6. What is SQLJ used for? Describe the two types of iterators available in SQLJ.

Exercises

10.7. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. Write a program segment to read a student's name and print his or her grade point average, assuming that A = 4, B = 3, C = 2, and D = 1 points. Use embedded SQL with C as the host language.

10.8. Repeat Exercise 10.7, but use SQLJ with Java as the host language.

10.9. Consider the library relational database schema in Figure 6.6. Write a program segment that retrieves the list of books that became overdue yesterday and that prints the book title and borrower name for each. Use embedded SQL with C as the host language.

10.10. Repeat Exercise 10.9, but use SQLJ with Java as the host language.

10.11. Repeat Exercises 10.7 and 10.9, but use SQL/CLI with C as the host language.

10.12. Repeat Exercises 10.7 and 10.9, but use JDBC with Java as the host language.

10.13. Repeat Exercise 10.7, but write a function in SQL/PSM.

10.14. Create a function in PSM that computes the median salary for the EMPLOYEE table shown in Figure 5.5.

Selected Bibliography

There are many books that describe various aspects of SQL database programming. For example, Sunderraman (2007) describes programming on the Oracle 10g DBMS and Reese (1997) focuses on JDBC and Java programming. Many Web resources are also available.

This page intentionally left blank

Web Database
Programming Using PHP

I n the previous chapter, we gave an overview of database programming techniques using traditional programming languages, and we used the Java and C programming languages in our examples. We now turn our attention to how databases are accessed from scripting languages. Many Internet applications that provide Web interfaces to access information stored in one or more databases use scripting languages. These languages are often used to generate HTML documents, which are then displayed by the Web browser for interaction with the user. In our presentation, we assume that the reader is familiar with basic HTML concepts.

Basic HTML is useful for generating *static* Web pages with fixed text and other objects, but most Internet applications require Web pages that provide interactive features with the user. For example, consider the case of an airline customer who wants to check the arrival time and gate information of a particular flight. The user may enter information such as a date and flight number in certain fields of the Web page. The Web interface will send this information to the application program, which formulates and submits a query to the airline database server to retrieve the information that the user needs. The database information is sent back to the Web page for display. Such Web pages, where part of the information is extracted from databases or other data sources, are called *dynamic* Web pages. The data extracted and displayed each time will be for different flights and dates.

There are various techniques for programming dynamic features into Web pages. We will focus on one technique here, which is based on using the PHP open source server side scripting language. PHP originally stood for Personal Home Page, but now stands for PHP Hypertext Processor. PHP has experienced widespread use. The interpreters for PHP are provided free of charge and are written in the C language so

they are available on most computer platforms. A PHP interpreter provides a Hypertext Preprocessor, which will execute PHP commands in a text file and create the desired HTML file. To access databases, a library of PHP functions needs to be included in the PHP interpreter, as we will discuss in Section 11.3. PHP programs are executed on the Web server computer. This is in contrast to some scripting languages, such as JavaScript, that are executed on the client computer. There are many other popular scripting languages that can be used to access databases and create dynamic Web pages, such as JavaScript, Ruby, Python, and PERL, to name a few.

This chapter is organized as follows. Section 11.1 gives a simple example to illustrate how PHP can be used. Section 11.2 gives a general overview of the PHP language and how it is used to program some basic functions for interactive Web pages. Section 11.3 focuses on using PHP to interact with SQL databases through a library of functions known as PEAR DB. Section 11.4 lists some of the additional technologies associated with Java for Web and database programming (we already discussed JDBC and SQLJ in Chapter 10). Finally, Section 11.5 contains a chapter summary.

11.1 A Simple PHP Example

PHP is an open source general-purpose scripting language. The interpreter engine for PHP is written in the C programming language so it can be used on nearly all types of computers and operating systems. PHP usually comes installed with the UNIX operating system. For computer platforms with other operating systems such as Windows, Linux, or Mac OS, the PHP interpreter can be downloaded from: http://www.php.net. As with other scripting languages, PHP is particularly suited for manipulation of text pages, and in particular for manipulating dynamic HTML pages at the Web server computer. This is in contrast to JavaScript, which is downloaded with the Web pages to execute on the client computer.

PHP has libraries of functions for accessing databases stored under various types of relational database systems such as Oracle, MySQL, SQLServer, and any system that supports the ODBC standard (see Chapter 10). Under the three-tier architecture (see Chapter 2), the DBMS would reside at the **bottom-tier database server**. PHP would run at the **middle-tier Web server**, where the PHP program commands would manipulate the HTML files to create the customized dynamic Web pages. The HTML is then sent to the **client tier** for display and interaction with the user.

Consider the PHP example shown in Figure 11.1(a), which prompts a user to enter the first and last name and then prints a welcome message to that user. The line numbers are not part of the program code; they are used below for explanation purposes only:

1. Suppose that the file containing PHP script in program segment P1 is stored in the following Internet location: http://www.myserver.com/example/greeting.php. Then if a user types this address in the browser, the PHP interpreter would start interpreting the code and produce the form shown in Figure 11.1(b). We will explain how that happens as we go over the lines in code segment P1.

(a)

```
   //Program Segment P1:
0) <?php
1) // Printing a welcome message if the user submitted their name
   // through the HTML form
2) if ($_POST['user_name']) {
3)   print("Welcome, ") ;
4)   print($_POST['user_name']);
5) }
6) else {
7)   // Printing the form to enter the user name since no name has
     // been entered yet
8)   print <<<_HTML_
9)   <FORM method="post" action="$_SERVER['PHP_SELF']">
10)   Enter your name: <input type="text" name="user_name">
11)   <BR/>
12)   <INPUT type="submit" value="SUBMIT NAME">
13)   </FORM>
14)   _HTML_;
15) }
16) ?>
```

(b)

Enter your name: [   ]

[SUBMIT NAME]

(c)

Enter your name: [John Smith]

[SUBMIT NAME]

(d)

Welcome, John Smith

Figure 11.1
(a) PHP program segment for entering a greeting.
(b) Initial form displayed by PHP program segment.
(c) User enters name *John Smith*. (d) Form prints
welcome message for *John Smith*.

2. Line 0 shows the PHP start tag <?php, which indicates to the PHP inter-preter engine that it should process all subsequent text lines until it encoun-ters the PHP end tag ?>, shown on line 16. Text outside of these tags is printed as is. This allows PHP code segments to be included within a larger HTML file. Only the sections in the file between <?php and ?> are processed by the PHP preprocessor.

3. Line 1 shows one way of posting comments in a PHP program on a single line started by //. Single-line comments can also be started with #, and end at the end of the line in which they are entered. Multiple-line comments start with /* and end with */.

4. The **auto-global** predefined PHP variable $_POST (line 2) is an array that holds all the values entered through form parameters. Arrays in PHP are

dynamic arrays, with no fixed number of elements. They can be numerically indexed arrays whose indexes (positions) are numbered (0, 1, 2, …), or they can be associative arrays whose indexes can be any string values. For example, an associative array indexed based on color can have the indexes {"red", "blue", "green"}. In this example, `$_POST` is associatively indexed by the name of the posted value `user_name` that is specified in the name attribute of the input tag on line 10. Thus `$_POST['user_name']` will contain the value typed in by the user. We will discuss PHP arrays further in Section 11.2.2.

5. When the Web page at http://www.myserver.com/example/greeting.php is first opened, the `if` condition in line 2 will evaluate to false because there is no value yet in `$_POST['user_name']`. Hence, the PHP interpreter will process lines 6 through 15, which create the text for an HTML file that displays the form shown in Figure 11.1(b). This is then displayed at the client side by the Web browser.

6. Line 8 shows one way of creating **long text strings** in an HTML file. We will discuss other ways to specify strings later in this section. All text between an opening `<<<_HTML_` and a closing `_HTML_;` is printed into the HTML file as is. The closing `_HTML_;` must be alone on a separate line. Thus, the text added to the HTML file sent to the client will be the text between lines 9 and 13. This includes HTML tags to create the form shown in Figure 11.1(b).

7. PHP **variable names** start with a $ sign and can include characters, numbers, and the underscore character _. The PHP auto-global (predefined) variable `$_SERVER` (line 9) is an array that includes information about the local server. The element `$_SERVER['PHP_SELF']` in the array is the path name of the PHP file currently being executed on the server. Thus, the action attribute of the form tag (line 9) instructs the PHP interpreter to reprocess the same file, once the form parameters are entered by the user.

8. Once the user types the name *John Smith* in the text box and clicks on the *SUBMIT NAME* button (Figure 11.1(c)), program segment *P1* is reprocessed. This time, `$_POST['user_name']` will include the string `"John Smith"`, so lines 3 and 4 will now be placed in the HTML file sent to the client, which displays the message in Figure 11.1(d).

As we can see from this example, a PHP program can create two different HTML commands depending on whether the user just started or whether they had already submitted their name through the form. In general, a PHP program can create numerous variations of HTML text in an HTML file at the server depending on the particular conditional paths taken in the program. Hence, the HTML sent to the client will be different depending on the interaction with the user. This is one way in which PHP is used to create *dynamic* Web pages.

11.2 Overview of Basic Features of PHP

In this section we give an overview of a few of the features of PHP that are useful in creating interactive HTML pages. Section 11.3 will focus on how PHP programs can access databases for querying and updating. We cannot give a comprehensive

discussion of PHP; there are many books that focus solely on PHP. Rather, we focus on illustrating certain features of PHP that are particularly suited for creating dynamic Web pages that contain database access commands. This section covers some PHP concepts and features that will be needed when we discuss database access in Section 11.3.

11.2.1 PHP Variables, Data Types, and Programming Constructs

PHP **variable names** start with the $ symbol and can include characters, letters, and the underscore character (_). No other special characters are permitted. Variable names are case sensitive, and the first character cannot be a number. Variables are not typed. The values assigned to the variables determine their type. In fact, the same variable can change its type once a new value is assigned to it. Assignment is via the = operator.

Since PHP is directed toward text processing, there are several different types of string values. There are also many functions available for processing strings. We only discuss some basic properties of string values and variables here. Figure 11.2 illustrates some string values. There are three main ways to express strings and text:

1. **Single-quoted strings.** Enclose the string between single quotes, as in lines 0, 1, and 2. If a single quote is needed within the string, use the escape character (\) (see line 2).

2. **Double-quoted strings.** Enclose strings between double quotes as in line 7. In this case, *variable names appearing within the string* are replaced by the values that are currently stored in these variables. The interpreter identifies variable names within double-quoted strings by their initial character $ and replaces them with the value in the variable. This is known as **interpolating variables** within strings. Interpolation does not occur in single-quoted strings.

3. **Here documents.** Enclose a part of a document between a <<<DOCNAME and end it with a single line containing the document name DOCNAME.

```
0) print 'Welcome to my Web site.';
1) print 'I said to him, "Welcome Home"';
2) print 'We\'ll now visit the next Web site';
3) printf('The cost is $%.2f and the tax is $%.2f',
   $cost, $tax) ;
4) print strtolower('AbCdE');
5) print ucwords(strtolower('JOHN smith'));
6) print 'abc' . 'efg'
7) print "send your email reply to: $email_address"
8) print <<<FORM_HTML
9) <FORM method="post" action="$_SERVER['PHP_SELF']">
10) Enter your name: <input type="text" name="user_name">
11) FORM_HTML
```

Figure 11.2

Illustrating basic PHP string and text values.

DOCNAME can be any string as long as it used both to start and end the here document. This is illustrated in lines 8 through 11 in Figure 11.2. Variables are also interpolated by replacing them with their string values if they appear inside here documents. This feature is used in a similar way to double-quoted strings, but it is more convenient for multiple-line text.

4. **Single and double quotes.** Single and double quotes used by PHP to enclose strings should be *straight* quotes ("") on both sides of the string. The text editor that creates these quotes should not produce *curly* opening and clos-ing quotes (" ") around the string.

There is also a string concatenate operator specified by the period (.) symbol, as illustrated in line 6 of Figure 11.2. There are many string functions. We only illus-trate a couple of them here. The function strtolower changes the alphabetic char-acters in the string to all lowercase, whereas the function ucwords capitalizes all the words in a string. These are illustrated in lines 4 and 5 in Figure 11.2.

The general rule is to use single-quoted strings for literal strings that contain no PHP program variables and the other two types (double-quoted strings and here documents) when the values from variables need to be interpolated into the string. For large blocks of multiline text, the program should use the *here documents* style for strings.

PHP also has numeric data types for integers and floating points and generally fol-lows the rules of the C programming language for processing these types. Numbers can be formatted for printing into strings by specifying the number of digits that follow the decimal point. A variation of the print function called printf (print formatted) allows formatting of numbers within a string, as illustrated in line 3 of Figure 11.2.

There are the standard programming language constructs of for-loops, while-loops, and conditional if-statements. They are generally similar to their C language coun-terparts. We will not discuss them here. Similarly, *any value* evaluates to true if used as a Boolean expression *except for* numeric zero (0) and blank string, which evalu-ate to false. There are also literal true and false values that can be assigned. The comparison operators also generally follow C language rules. They are == (equal), != (not equal), > (greater than), >= (greater than or equal), < (less than), and <= (less than or equal).

11.2.2 PHP Arrays

Arrays are very important in PHP, since they allow lists of elements. They are used frequently in forms that employ pull-down menus. A single-dimensional array is used to hold the list of choices in the pull-down menu. For database query results, two-dimensional arrays are used, with the first dimension representing *rows* of a table and the second dimension representing *columns* (attributes) within a row. There are two main types of arrays: numeric and associative. We discuss each of these in the context of single-dimensional arrays next.

A **numeric array** associates a numeric index (or position or sequence number) with each element in the array. Indexes are integer numbers that start at zero and grow incrementally. An element in the array is referenced through its index. An **associative array** provides pairs of (key => value) elements. The value of an element is referenced through its key, and all key values in a particular array must be unique. The element values can be strings or integers, or they can be arrays themselves, thus leading to higher dimensional arrays.

Figure 11.3 gives two examples of array variables: $teaching and $courses. The first array $teaching is associative (see line 0 in Figure 11.3), and each element associates a course name (as key) with the name of the course instructor (as value). There are three elements in this array. Line 1 shows how the array may be updated. The first command in line 1 assigns a new instructor to the course 'Graphics' by updating its value. Since the key value 'Graphics' already exists in the array, no new element is created but the existing value is updated. The second command creates a new element since the key value 'Data Mining' did not exist in the array before. New elements are added at the end of the array.

If we only provide values (no keys) as array elements, the keys are automatically numeric and numbered 0, 1, 2, … . This is illustrated in line 5 of Figure 11.3, by the $courses array. Both associative and numeric arrays have no size limits. If some value of another data type, say an integer, is assigned to a PHP variable that was holding an array, the variable now holds the integer value and the array contents are lost. Basically, most variables can be assigned to values of any data type at any time.

There are several different techniques for looping through arrays in PHP. We illustrate two of these techniques in Figure 11.3. Lines 3 and 4 show one method of looping through all the elements in an array using the `foreach` construct, and printing the key and value of each element on a separate line. Lines 7 through 10 show how a traditional for-loop construct can be used. A built-in function count

---

Figure 11.3
Illustrating basic PHP array processing.

```
 0) $teaching = array('Database' => 'Smith', 'OS' => 'Carrick',
                      'Graphics' => 'Kam');
 1) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Li';
 2) sort($teaching);
 3) foreach ($teaching as $key => $value) {
 4)   print " $key : $value\n";}
 5) $courses = array('Database', 'OS', 'Graphics', 'Data Mining');
 6) $alt_row_color = array('blue', 'yellow');
 7) for ($i = 0, $num = count($courses); i < $num; $i++) {
 8)   print '<TR bgcolor="' . $alt_row_color[$i % 2] . '">';
 9)   print "<TD>Course $i is</TD><TD>$course[$i]</TD></TR>\n";
10) }
```

(line 7) returns the current number of elements in the array, which is assigned to the variable $num and used to control ending the loop.

The example in lines 7 through 10 also illustrates how an HTML table can be displayed with alternating row colors, by setting the two colors in an array $alt_row_color (line 8). Each time through the loop, the remainder function $i % 2 switches from one row (index 0) to the next (index 1) (see line 8). The color is assigned to the HTML *bgcolor* attribute of the <TR> (table row) tag.

The count function (line 7) returns the current number of elements in the array. The sort function (line 2) sorts the array based on the element values in it (not the keys). For associative arrays, each key remains associated with the same element value after sorting. This does not occur when sorting numeric arrays. There are many other functions that can be applied to PHP arrays, but a full discussion is outside the scope of our presentation.

11.2.3 PHP Functions

As with other programming languages, **functions** can be defined in PHP to better structure a complex program and to share common sections of code that can be reused by multiple applications. The newer version of PHP, PHP5, also has object-oriented features, but we will not discuss these here because we are focusing on the basics of PHP. Basic PHP functions can have arguments that are *passed by value*. Global variables can be accessed within functions. Standard scope rules apply to variables that appear within a function and within the code that calls the function.

We now give two simple examples to illustrate basic PHP functions. In Figure 11.4, we show how we could rewrite the code segment P1 from Figure 11.1(a) using functions. The code segment P1′ in Figure 11.4 has two functions: display_welcome() (lines 0 to 3) and display_empty_form() (lines 5 to 13). Neither of these functions has arguments; nor do they have return values. Lines 14 through 19 show how we can call these functions to produce the same effect as the segment of code P1 in Figure 11.1(a). As we can see in this example, functions can be used just to make the PHP code better structured and easier to follow.

A second example is shown in Figure 11.5. Here we are using the $teaching array introduced in Figure 11.3. The function course_instructor() in lines 0 to 8 in Figure 11.5 has two arguments: $course (a string holding a course name) and $teaching_assignments (an associative array holding course assignments, similar to the $teaching array shown in Figure 11.3). The function finds the name of the instructor who teaches a particular course. Lines 9 to 14 in Figure 11.5 show how this function may be used.

The function call in line 11 would return the string: *Smith is teaching Database*, because the array entry with the key 'Database' has the value 'Smith' for instructor. On the other hand, the function call on line 13 would return the string: *there is no Computer Architecture course* because there is no entry in the array with the key

Figure 11.4
Rewriting program segment P1 as P1′ using functions.

```
    //Program Segment P1′:
 0) function display_welcome() {
 1)     print("Welcome, ") ;
 2)     print($_POST['user_name']);
 3) }
 4)
 5) function display_empty_form(); {
 6) print <<<_HTML_
 7) <FORM method="post" action="$_SERVER['PHP_SELF']">
 8) Enter your name: <INPUT type="text" name="user_name">
 9) <BR/>
10) <INPUT type="submit" value="Submit name">
11) </FORM>
12) _HTML_;
13) }
14) if ($_POST['user_name']) {
15)   display_welcome();
16) }
17) else {
18)   display_empty_form();
19) }
```

Figure 11.5
Illustrating a function with arguments and return value.

```
 0) function course_instructor ($course, $teaching_assignments) {
 1)   if (array_key_exists($course, $teaching_assignments)) {
 2)   $instructor = $teaching_assignments[$course];
 3)   RETURN "$instructor is teaching $course";
 4)   }
 5)   else {
 6)   RETURN "there is no $course course";
 7)   }
 8) }
 9) $teaching = array('Database' => 'Smith', 'OS' => 'Carrick',
                      'Graphics' => 'Kam');
10) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Li';
11) $x = course_instructor('Database', $teaching);
12) print($x);
13) $x = course_instructor('Computer Architecture', $teaching);
14) print($x);
```

'Computer Architecture'. A few comments about this example and about PHP functions in general:

- The built-in PHP array function `array_key_exists($k, $a)` returns true if the value in variable `$k` *exists as a key* in the *associative array* in the variable `$a`. In our example, it checks whether the `$course` value provided exists as a key in the array `$teaching_assignments` (line 1 in Figure 11.5).

- Function arguments are passed by value. Hence, in this example, the calls in lines 11 and 13 could not change the array `$teaching` provided as argument for the call. The values provided in the arguments are passed (copied) to the function arguments when the function is called.

- Return values of a function are placed after the RETURN keyword. A function can return any type. In this example, it returns a string type. Two different strings can be returned in our example, depending on whether the `$course` key value provided exists in the array or not.

- Scope rules for variable names apply as in other programming languages. Global variables outside of the function cannot be used unless they are referred to using the built-in PHP array `$GLOBALS`. Basically, `$GLOBALS['abc']` will access the value in a global variable `$abc` defined outside the function. Otherwise, variables appearing inside a function are local even if there is a global variable with the same name.

The previous discussion gives a brief overview of PHP functions. Many details are not discussed since it is not our goal to present PHP in detail.

11.2.4 PHP Server Variables and Forms

There are a number of built-in entries in a PHP auto-global built-in array variable called `$_SERVER` that can provide the programmer with useful information about the server where the PHP interpreter is running, as well as other information. These may be needed when constructing the text in an HTML document (for example, see line 7 in Figure 11.4). Here are some of these entries:

1. `$_SERVER['SERVER_NAME']`. This provides the Web site name or the Uniform Resource Locator (URL) of the server computer where the PHP interpreter is running. For example, if the PHP interpreter is running on the Web site http://www.uta.edu, then this string would be the value in `$_SERVER['SERVER_NAME']`.

2. `$_SERVER['REMOTE_ADDRESS']`. This is the IP (Internet Protocol) address of the client user computer that is accessing the server; for example, 129.107.61.8.

3. `$_SERVER['REMOTE_HOST']`. This is the Web site name (URL) of the client user computer; for example, abc.uta.edu. In this case, the server will need to translate the name into an IP address to access the client.

4. `$_SERVER['PATH_INFO']`. This is the part of the URL address that comes after a backslash (/) at the end of the URL.

5. **$_SERVER['QUERY_STRING'].** This provides the string that holds parameters in a URL after a question mark (?) at the end of the URL. This can hold search parameters, for example.

6. **$_SERVER['DOCUMENT_ROOT'].** This is the root directory that holds the files on the Web server that are accessible to client users.

These and other entries in the $_SERVER array are usually needed when creating the HTML file to be sent to the client for display.

Another important PHP auto-global built-in array variable is called $_POST. This provides the programmer with input values submitted by the user through HTML forms specified in the HTML <INPUT> tag and other similar tags. For example, in Figure 11.4, line 14, the variable $_POST['user_name'] provides the programmer with the value typed in by the user in the HTML form specified via the <INPUT> tag on line 8 in Figure 11.4. The keys to this array are the names of the various input parameters provided via the form, for example by using the name attribute of the HTML <INPUT> tag as on line 8. When users enter data through forms, the data values are stored in this array.

11.3 Overview of PHP Database Programming

There are various techniques for accessing a database through a programming language. We discussed some of the techniques in Chapter 10, in the overviews of how to access an SQL database using the C and Java programming languages. In particular, we discussed embedded SQL, JDBC, SQL/CLI (similar to ODBC), and SQLJ. In this section we give an overview of how to access the database using the script language PHP, which is suitable for creating Web interfaces for searching and updating databases, as well as dynamic Web pages.

There is a PHP database access function library that is part of PHP Extension and Application Repository (PEAR), which is a collection of several libraries of functions for enhancing PHP. The PEAR DB library provides functions for database access. Many database systems can be accessed from this library, including Oracle, MySQL, SQLite, and Microsoft SQLServer, among others.

We will discuss several functions that are part of PEAR DB in the context of some examples. Section 11.3.1 shows how to connect to a database using PHP. Section 11.3.2 discusses how data collected from HTML forms can be used to insert a new record in a database table. Section 11.3.3 shows how retrieval queries can be executed and have their results displayed within a dynamic Web page.

11.3.1 Connecting to a Database

To use the database functions in a PHP program, the PEAR DB library module called DB.php must be loaded. In Figure 11.6, this is done in line 0 of the example. The DB library functions can now be accessed using DB::<function_name>. The function for connecting to a database is called DB::connect('string'),

```
 0) require 'DB.php';
 1) $d = DB::connect('oci8://acct1:pass12@www.host.com/db1');
 2) if (DB::isError($d)) { die("cannot connect - " . $d->getMessage());}
      ...
 3) $q = $d->query("CREATE TABLE EMPLOYEE
 4)    (Emp_id INT,
 5)    Name VARCHAR(15),
 6)    Job VARCHAR(10),
 7)    Dno INT);" );
 8) if (DB::isError($q)) { die("table creation not successful - " .
                          $q->getMessage()); }
      ...
 9) $d->setErrorHandling(PEAR_ERROR_DIE);
      ...
10) $eid = $d->nextID('EMPLOYEE');
11) $q = $d->query("INSERT INTO EMPLOYEE VALUES
12)    ($eid, $_POST['emp_name'], $_POST['emp_job'], $_POST['emp_dno'])" );
      ...
13) $eid = $d->nextID('EMPLOYEE');
14) $q = $d->query('INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)',
15) array($eid, $_POST['emp_name'], $_POST['emp_job'], $_POST['emp_dno']) );
```

Figure 11.6
Connecting to a database, creating a table, and inserting a record.

where the string argument specifies the database information. The format for 'string' is:

> <DBMS software>://<user account>:<password>@<database server>

In Figure 11.6, line 1 connects to the database that is stored using Oracle (specified via the string oci8). The <DBMS software> portion of the 'string' specifies the particular DBMS software package being connected to. Some of the DBMS software packages that are accessible through PEAR DB are:

- ■ **MySQL.** Specified as mysql for earlier versions and mysqli for later versions starting with version 4.1.2.
- ■ **Oracle.** Specified as oc8i for versions 7, 8, and 9. This is used in line 1 of Figure 11.6.
- ■ **SQLite.** Specified as sqlite.
- ■ **Microsoft SQL Server.** Specified as mssql.
- ■ **Mini SQL.** Specified as msql.
- ■ **Informix.** Specified as ifx.
- ■ **Sybase.** Specified as sybase.
- ■ **Any ODBC-compliant system.** Specified as odbc.

The above is not a comprehensive list.

Following the `<DB software>` in the string argument passed to `DB::connect` is the separator `://` followed by the user account name `<user account>` followed by the separator `:` and the account password `<password>`. These are followed by the separator `@` and the server name and directory `<database server>` where the database is stored.

In line 1 of Figure 11.6, the user is connecting to the server at www.host.com/db1 using the account name `acct1` and password `pass12` stored under the Oracle DBMS `oci8`. The whole string is passed using `DB::connect`. The connection information is kept in the database connection variable `$d`, which is used whenever an operation to this particular database is applied.

Checking for errors. Line 2 in Figure 11.6 shows how to check whether the connection to the database was established successfully or not. PEAR DB has a function `DB::isError`, which can determine whether any database access operation was successful or not. The argument to this function is the database connection variable (`$d` in this example). In general, the PHP programmer can check after every database call to determine whether the last database operation was successful or not, and terminate the program (using the *die* function) if it was not successful. An error message is also returned from the database via the operation `$d->get_message()`. This can also be displayed as shown in line 2 of Figure 11.6.

Submitting queries and other SQL statements. In general, most SQL commands can be sent to the database once a connection is established by using the *query* function. The function `$d->query` takes an SQL command as its string argument and sends it to the database server for execution. In Figure 11.6, lines 3 to 7 send a `CREATE TABLE` command to create a table called EMPLOYEE with four attributes. Whenever a query or SQL statement is executed, the result of the query is assigned to a query variable, which is called `$q` in our example. Line 8 checks whether the query was executed successfully or not.

The PHP PEAR DB library offers an alternative to having to check for errors after every database command. The function

```
$d->setErrorHandling(PEAR_ERROR_DIE)
```

will terminate the program and print the default error messages if any subsequent errors occur when accessing the database through connection `$d` (see line 9 in Figure 11.6).

11.3.2 Collecting Data from Forms and Inserting Records

It is common in database applications to collect information through HTML or other types of Web forms. For example, when purchasing an airline ticket or applying for a credit card, the user has to enter personal information such as name, address, and phone number. This information is typically collected and stored in a database record on a database server.

Lines 10 through 12 in Figure 11.6 illustrate how this may be done. In this example, we omitted the code for creating the form and collecting the data, which can be a variation of the example in Figure 11.1. We assume that the user entered valid values in the input parameters called emp_name, emp_job, and emp_dno. These would be accessible via the PHP auto-global array $_POST as discussed at the end of Section 11.2.4.

In the SQL INSERT command shown on lines 11 and 12 in Figure 11.6, the array entries $POST['emp_name'], $POST['emp_job'], and $POST['emp_dno'] will hold the values collected from the user through the input form of HTML. These are then inserted as a new employee record in the EMPLOYEE table.

This example also illustrates another feature of PEAR DB. It is common in some applications to create a unique record identifier for each new record inserted into the database.[1]

PHP has a function $d–>nextID to create a sequence of unique values for a particular table. In our example, the field Emp_id of the EMPLOYEE table (see Figure 11.6, line 4) is created for this purpose. Line 10 shows how to retrieve the next unique value in the sequence for the EMPLOYEE table and insert it as part of the new record in lines 11 and 12.

The code for insert in lines 10 to 12 in Figure 11.6 may allow malicious strings to be entered that can alter the INSERT command. A safer way to do inserts and other queries is through the use of **placeholders** (specified by the ? symbol). An example is illustrated in lines 13 to 15, where another record is to be inserted. In this form of the $d–>query() function, there are two arguments. The first argument is the SQL statement, with one or more ? symbols (placeholders). The second argument is an array, whose element values will be used to replace the placeholders in the order they are specified (see lines 13 to 15 in Figure 11.6).

11.3.3 Retrieval Queries from Database Tables

We now give three examples of retrieval queries through PHP, shown in Figure 11.7. The first few lines 0 to 3 establish a database connection $d and set the error handling to the default, as we discussed in the previous section. The first query (lines 4 to 7) retrieves the name and department number of all employee records. The query variable $q is used to refer to the **query result**. A while-loop to go over each row in the result is shown in lines 5 to 7. The function $q–>fetchRow() in line 5 serves to retrieve the next record in the query result and to control the loop. The looping starts at the first record.

The second query example is shown in lines 8 to 13 and illustrates a dynamic query. In this query, the conditions for selection of rows are based on values input by the user. Here we want to retrieve the names of employees who have a

---

[1]This would be similar to the system-generated OID discussed in Chapter 12 for object and object-relational database systems.

```
 0) require 'DB.php';
 1) $d = DB::connect('oci8://acct1:pass12@www.host.com/dbname');
 2) if (DB::isError($d)) { die("cannot connect - " . $d->getMessage()); }
 3) $d->setErrorHandling(PEAR_ERROR_DIE);
    ...
 4) $q = $d->query('SELECT Name, Dno FROM EMPLOYEE');
 5) while ($r = $q->fetchRow()) {
 6)   print "employee $r[0] works for department $r[1] \n" ;
 7) }
    ...
 8) $q = $d->query('SELECT Name FROM EMPLOYEE WHERE Job = ? AND Dno = ?',
 9)   array($_POST['emp_job'], $_POST['emp_dno']) );
10) print "employees in dept $_POST['emp_dno'] whose job is
        $_POST['emp_job']: \n"
11) while ($r = $q->fetchRow()) {
12)   print "employee $r[0] \n" ;
13) }
    ...
14) $allresult = $d->getAll('SELECT Name, Job, Dno FROM EMPLOYEE');
15) foreach ($allresult as $r) {
16)   print "employee $r[0] has job $r[1] and works for department $r[2] \n" ;
17) }
    ...
```

Figure 11.7
Illustrating database retrieval queries.

specific job and work for a particular department. The particular job and department number are entered through a form in the array variables $POST['emp_job'] and $POST['emp_dno']. If the user had entered 'Engineer' for the job and 5 for the department number, the query would select the names of all engineers who worked in department 5. As we can see, this is a dynamic query whose results differ depending on the choices that the user enters as input. We used two ? placeholders in this example, as discussed at the end of Section 11.3.2.

The last query (lines 14 to 17) shows an alternative way of specifying a query and looping over its rows. In this example, the function $d=>getAll holds all the records in a query result in a single variable, called $allresult. To loop over the individual records, a foreach loop can be used, with the row variable $r iterating over each row in $allresult.[2]

As we can see, PHP is suited for both database access and creating dynamic Web pages.

_____

[2]The $r variable is similar to the cursors and iterator variables discussed in Chapters 10 and 12.

11.4 Brief Overview of Java Technologies for Database Web Programming

The parts of the PHP scripting language that we discussed run on the application server and serve as a conduit that collects client user input through forms, formulates database queries and submits them to the database server, and then creates dynamic HTML Web pages to display query results. The Java environment has components that run on the server and other components that can run on the client machine. It also has standards for exchanging data objects. We briefly discuss some of these components here that are related to Web and database access. We already discussed JDBC and SQLJ in some detail in Chapter 10.

Java Servlets. Servlets are Java objects that can reside on the Web server machine and manage interactions with the client. They can store information that was submitted by the client during a session, so that this information can be used to generate database queries. Servlet objects can also store query results so that parts of these results can be formatted as HTML and sent to the client for display. The servlet object can maintain all the information produced during a particular client interaction until the client session is terminated.

Java Server Pages (JSP). This allows scripting at the server to produce dynamic Web pages to be sent at the client in a manner somewhat similar to PHP. However, it is associated with the Java language and the scripting can be combined with Java code.

JavaScript. JavaScript is a scripting language that is different from the Java programming language and was developed separately. It is widely used in Web applications, and it can run on the client computer or on the server.

Java Script Object Notation (JSON). This is a text-based representation of data objects, so that data can be formatted in JSON and exchanged between clients and servers over the Web in text format. It can be considered as an alternative to XML (see Chapter 13) and represents objects using attribute-value pairs. JSON has also been adopted as the data model by some newer database systems known as NOSQL systems, such as MongoDB (see Chapter 24).

11.5 Summary

In this chapter, we gave an overview of how to convert some structured data from databases into elements to be entered or displayed on a Web page. We focused on the PHP scripting language, which is becoming very popular for Web database programming. Section 11.1 presented some PHP basics for Web programming through a simple example. Section 11.2 gave some of the basics of the PHP language, including its array and string data types that are used extensively. Section 11.3 presented an overview of how PHP can be used to specify various types of database commands, including creating tables, inserting new records, and retrieving database records. PHP runs at the server computer in comparison to some other scripting languages that run on the client computer. Section 11.4 introduced some of the technologies associated with Java that can be used in similar contexts.

We gave only a very basic introduction to PHP. There are many books as well as many Web sites devoted to introductory and advanced PHP programming. Many libraries of functions also exist for PHP, as it is an open source product.

Review Questions

11.1. Why are scripting languages popular for programming Web applications? Where in the three-tier architecture does a PHP program execute? Where does a JavaScript program execute?

11.2. What type of programming language is PHP?

11.3. Discuss the different ways of specifying strings in PHP.

11.4. Discuss the different types of arrays in PHP.

11.5. What are PHP auto-global variables? Give some examples of PHP auto-global arrays, and discuss how each is typically used.

11.6. What is PEAR? What is PEAR DB?

11.7. Discuss the main functions for accessing a database in PEAR DB, and how each is used.

11.8. Discuss the different ways for looping over a query result in PHP.

11.9. What are placeholders? How are they used in PHP database programming?

Exercises

11.10. Consider the LIBRARY database schema shown in Figure 4.6. Write PHP code to create the tables of this schema.

11.11. Write a PHP program that creates Web forms for entering the information about a new BORROWER entity. Repeat for a new BOOK entity.

11.12. Write PHP Web interfaces for the queries specified in Exercise 6.18.

Selected Bibliography

There are many sources for PHP programming, both in print and on the Web. We give two books as examples. A very good introduction to PHP is given in Sklar (2005). For advanced Web site development, the book by Schlossnagle (2005) provides many detailed examples. Nixon (2014) has a popular book on web programming that covers PHP, Javascript, Jquery, CSS and HTML5.

This page intentionally left blank

part **5**

Object, Object-Relational, and XML: Concepts, Models, Languages, and Standards

This page intentionally left blank

Object and Object-Relational Databases

In this chapter, we discuss the features of object-oriented data models and show how some of these features have been incorporated in relational database systems and the SQL standard. Some features of object data models have also been incorporated into the data models of newer types of database systems, known as NOSQL systems (see Chapter 24). In addition, the XML model (see Chapter 13) has similarities to the object model. So an introduction to the object model will give a good perspective on many of the recent advances in database technology. Database systems that were based on the object data model were known originally as object-oriented databases (OODBs) but are now referred to as **object databases** (**ODBs**).Traditional data models and systems, such as network, hierarchical, and relational have been quite successful in developing the database technologies required for many traditional business database applications. However, they have certain shortcomings when more complex database applications must be designed and implemented—for example, databases for engineering design and manufacturing (CAD/CAM and CIM[1]), biological and other sciences, telecommunications, geographic information systems, and multimedia.[2] These ODBs were developed for applications that have requirements requiring more complex structures for stored objects. A key feature of object databases is the power they give the designer to specify both the *structure* of complex objects and the *operations* that can be applied to these objects.

---

[1]Computer-aided design/computer-aided manufacturing and computer-integrated manufacturing.

[2]Multimedia databases must store various types of multimedia objects, such as video, audio, images, graphics, and documents (see Chapter 26).

Another reason for the creation of object-oriented databases is the vast increase in the use of object-oriented programming languages for developing software applications. Databases are fundamental components in many software systems, and traditional databases are sometimes difficult to use with software applications that are developed in an object-oriented programming language such as C++ or Java. Object databases are designed so they can be directly—or *seamlessly*—integrated with software that is developed using object-oriented programming languages.

Relational DBMS (RDBMS) vendors have also recognized the need for incorporating features that were proposed for object databases, and newer versions of relational systems have incorporated many of these features. This has led to database systems that are characterized as *object-relational* or ORDBMSs. A recent version of the SQL standard (2008) for RDBMSs, known as SQL/Foundation, includes many of these features, which were originally known as SQL/Object and have now been merged into the main SQL specification.

Although many experimental prototypes and commercial object-oriented database systems have been created, they have not found widespread use because of the popularity of relational and object-relational systems. The experimental prototypes included the Orion system developed at MCC, OpenOODB at Texas Instruments, the Iris system at Hewlett-Packard laboratories, the Ode system at AT&T Bell Labs, and the ENCORE/ObServer project at Brown University. Commercially available systems included GemStone Object Server of GemStone Systems, ONTOS DB of Ontos, Objectivity/DB of Objectivity Inc., Versant Object Database and FastObjects by Versant Corporation (and Poet), ObjectStore of Object Design, and Ardent Database of Ardent.

As commercial object DBMSs became available, the need for a standard model and language was recognized. Because the formal procedure for approval of standards normally takes a number of years, a consortium of object DBMS vendors and users, called ODMG, proposed a standard whose current specification is known as the ODMG 3.0 standard.

Object-oriented databases have adopted many of the concepts that were developed originally for object-oriented programming languages.[3] In Section 12.1, we describe the key concepts utilized in many object database systems and that were later incorporated into object-relational systems and the SQL standard. These include *object identity*, *object structure* and *type constructors*, *encapsulation of operations*, and the definition of *methods* as part of class declarations, mechanisms for storing objects in a database by making them *persistent,* and *type and class hierarchies* and *inheritance.* Then, in Section 12.2 we see how these concepts have been incorporated into the latest SQL standards, leading to object-relational databases. Object features were originally introduced in SQL:1999, and then updated in SQL:2008. In Section 12.3, we turn our attention to "pure" object database standards by presenting features of the object database standard ODMG 3.0 and the object definition

---

[3]Similar concepts were also developed in the fields of semantic data modeling and knowledge representation.

language ODL. Section 12.4 presents an overview of the database design process for object databases. Section 12.5 discusses the object query language (OQL), which is part of the ODMG 3.0 standard. In Section 12.6, we discuss programming language bindings, which specify how to extend object-oriented programming languages to include the features of the object database standard. Section 12.7 summarizes the chapter. Sections 12.3 through 12.6 may be left out if a less thorough introduction to object databases is desired.

12.1 Overview of Object Database Concepts

12.1.1 Introduction to Object-Oriented Concepts and Features

The term *object-oriented*—abbreviated *OO* or *O-O*—has its origins in OO programming languages, or OOPLs. Today OO concepts are applied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general. OOPLs have their roots in the SIMULA language, which was proposed in the late 1960s. The programming language Smalltalk, developed at Xerox PARC[4] in the 1970s, was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a *pure* OO programming language, meaning that it was explicitly designed to be object-oriented. This contrasts with *hybrid* OO programming languages, which incorporate OO concepts into an already existing language. An example of the latter is C++, which incorporates OO concepts into the popular C programming language.

An **object** typically has two components: state (value) and behavior (operations). It can have a *complex data structure* as well as *specific operations* defined by the programmer.[5] Objects in an OOPL exist only during program execution; therefore, they are called *transient objects.* An OO database can extend the existence of objects so that they are stored permanently in a database, and hence the objects become *persistent objects* that exist beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store persistent objects permanently in secondary storage and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms to efficiently locate the objects, concurrency control to allow object sharing among concurrent programs, and recovery from failures. An OO database system will typically interface with one or more OO programming languages to provide persistent and shared object capabilities.

The internal structure of an object in OOPLs includes the specification of **instance variables**, which hold the values that define the internal state of the object. An instance variable is similar to the concept of an *attribute* in the relational model,

---

[4]Palo Alto Research Center, Palo Alto, California.

[5]Objects have many other characteristics, as we discuss in the rest of this chapter.

except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some OO models insist that all operations a user can apply to an object must be pre-defined. This forces a *complete encapsulation* of objects. This rigid approach has been relaxed in most OO data models for two reasons. First, database users often need to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.

To encourage encapsulation, an operation is defined in two parts. The first part, called the *signature* or *interface* of the operation, specifies the operation name and arguments (or parameters). The second part, called the *method* or *body,* specifies the *implementation* of the operation, usually written in some general-purpose programming language. Operations can be invoked by passing a *message* to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence (see Chapter 2).

Another key concept in OO systems is that of type and class hierarchies and *inheritance.* This permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes. This makes it easier to develop the data types of a system incrementally and to *reuse* existing type definitions when creating new types of objects.

One problem in early OO database systems involved representing *relationships* among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships because it is useful to identify these relationships and make them visible to users. The ODMG object database standard has recognized this need and it explicitly represents binary relationships via a pair of *inverse references*, as we will describe in Section 12.3.

Another OO concept is *operator overloading,* which refers to an operation's ability to be applied to different types of objects; in such a situation, an *operation name* may refer to several distinct *implementations,* depending on the type of object it is applied to. This feature is also called *operator polymorphism.* For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle. This may require the use of *late binding* of the operation name to the appropriate method at runtime, when the type of object to which the operation is applied becomes known.

In the next several sections, we discuss in some detail the main characteristics of object databases. Section 12.1.2 discusses object identity; Section 12.1.3 shows how the types for complex-structured objects are specified via type constructors; Section 12.1.4 discusses encapsulation and persistence; and Section 12.1.5 presents inheritance concepts. Section 12.1.6 discusses some additional OO concepts, and Section 12.1.7 gives a summary of all the OO concepts that we introduced. In Section 12.2, we show how some of these concepts have been incorporated into the SQL:2008 standard for relational databases. Then in Section 12.3, we show how these concepts are realized in the ODMG 3.0 object database standard.

12.1.2 Object Identity, and Objects versus Literals

One goal of an ODB is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, a **unique identity** is assigned to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated **object identifier (OID)**. The value of an OID may not be visible to the external user but is used internally by the system to identify each object uniquely and to create and manage interobject references. The OID can be assigned to program variables of the appropriate type when needed.

The main property required of an OID is that it be **immutable**; that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an ODMS must have some mechanism for generating OIDs and preserving the immutability property. It is also desirable that each OID be used only once; that is, even if an object is removed from the database, its OID should not be assigned to another object. These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected. We can compare this with the relational model, where each relation must have a primary key attribute whose value identifies each tuple uniquely. If the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same real-world object (for example, using the Emp_id of an EMPLOYEE in one relation and the Ssn in another).

It is also inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database. However, some early ODMSs have used the physical address as the OID to increase the efficiency of object retrieval. If the physical address of the object changes, an *indirect pointer* can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage.

Some early OO data models required that everything—from a simple value to a complex object—was represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two identical basic values to have different OIDs, which can be useful in some cases. For example, the integer value 50 can sometimes be used to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would have the integer 50 as their value. Although useful as a theoretical model, this is not very practical, since it leads to the generation of too many OIDs. Hence, most ODBs allow for the representation of both objects and **literals** (or values). Every object must have an immutable OID, whereas a literal value has no OID and its value just stands for itself. Thus, a literal value is typically stored within an object and *cannot be referenced* from other objects. In many systems, complex structured literal values can also be created without having a corresponding OID if needed.

12.1.3 Complex Type Structures for Objects and Literals

Another feature of ODBs is that objects and literals may have a *type structure* of *arbitrary complexity* in order to contain all of the necessary information that describes the object or literal. In contrast, in traditional database systems, information about a complex object is often *scattered* over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation. In ODBs, a complex type may be constructed from other types by *nesting* of **type constructors**. The three most basic constructors are atom, struct (or tuple), and collection.

1. One type constructor has been called the **atom** constructor, although this term is not used in the latest object standard. This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating-point numbers, enumerated types, Booleans, and so on. These basic data types are called **single-valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.

2. A second type constructor is referred to as the **struct** (or **tuple**) constructor. This can create standard structured types, such as the tuples (record types) in the basic relational model. A structured type is made up of several components and is also sometimes referred to as a *compound* or *composite* type. More accurately, the struct constructor is not considered to be a type, but rather a **type generator**, because many different structured types can be created. For example, two different structured types that can be created are: struct Name<FirstName: string, MiddleInitial: char, LastName: string>, and struct CollegeDegree<Major: string, Degree: string, Year: date>. To create complex nested type structures in the object model, the *collection* type constructors are needed, which we discuss next. Notice that the type constructors *atom* and *struct* are the only ones available in the original (basic) relational model.

3. **Collection** (or *multivalued*) type constructors include the **set(T)**, **list(T)**, **bag(T)**, **array(T)**, and **dictionary(K,T)** type constructors. These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be **type generators** because many different types can be created. For example, set(*string*), set(*integer*), and set(*Employee*) are three different types that can be created from the *set* type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set(*string*) must be string values.

The *atom constructor* is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly. The *tuple constructor* can create structured values and objects of the form $<a_1:i_1, a_2:i_2, \dots, a_n:i_n>$, where each a_j is an attribute name[6] and each i_j is a value or an OID.

The other commonly used constructors are collectively referred to as collection types but have individual differences among them. The **set constructor** will create objects or literals that are a set of *distinct* elements $\{i_1, i_2, \dots, i_n\}$, all of the same type. The **bag constructor** (also called a *multiset*) is similar to a set except that the elements in a bag *need not be distinct*. The **list constructor** will create an *ordered list* $[i_1, i_2, \dots, i_n]$ of OIDs or values of the same type. A list is similar to a **bag** except that the elements in a list are *ordered,* and hence we can refer to the first, second, or *j*th element. The **array constructor** creates a single-dimensional array of elements of the same type. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size. Finally, the **dictionary constructor** creates a collection of key-value pairs (K, V), where the value of a key K can be used to retrieve the corresponding value V.

The main characteristic of a collection type is that its objects or values will be a *collection of objects or values of the same type* that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The **tuple** type constructor is often called a **structured type**, since it corresponds to the **struct** construct in the C and C++ programming languages.

An **object definition language** (**ODL**)[7] that incorporates the preceding type constructors can be used to define the object types for a particular database application. In Section 12.3 we will describe the standard ODL of ODMG, but first we introduce the concepts gradually in this section using a simpler notation. The type constructors can be used to define the *data structures* for an OO *database schema.* Figure 12.1 shows how we may declare EMPLOYEE and DEPARTMENT types.

In Figure 12.1, the attributes that refer to other objects—such as Dept of EMPLOYEE or Projects of DEPARTMENT—are basically OIDs that serve as **references** to other objects to represent *relationships* among the objects. For example, the attribute Dept

---

[6]Also called an *instance variable name* in OO terminology.

[7]This corresponds to the DDL (data definition language) of the database system (see Chapter 2).

```
            define type EMPLOYEE
                tuple  (   Fname:          string;
                           Minit :         char;
                           Lname:          string;
                           Ssn:            string;
                           Birth_date:     DATE;
                           Address:        string;
                           Sex:            char;
                           Salary:         float;
                           Supervisor:     EMPLOYEE;
                           Dept:           DEPARTMENT;
            define type DATE
                tuple  (   Year:           integer;
                           Month:          integer;
                           Day:            integer; );
            define type DEPARTMENT
                tuple  (   Dname:          string;
                           Dnumber:        integer;
                           Mgr:            tuple  (   Manager:       EMPLOYEE;
                                                      Start_date:    DATE; );
                           Locations:      set(string);
                           Employees:      set(EMPLOYEE);
                           Projects:       set(PROJECT); );
```

Figure 12.1
Specifying the object
types EMPLOYEE,
DATE, and
DEPARTMENT using
type constructors.

of EMPLOYEE is of type DEPARTMENT and hence is used to refer to a specific
DEPARTMENT object (the DEPARTMENT object where the employee works). The
value of such an attribute would be an OID for a specific DEPARTMENT object. A
binary relationship can be represented in one direction, or it can have an *inverse
reference.* The latter representation makes it easy to traverse the relationship in both
directions. For example, in Figure 12.1 the attribute Employees of DEPARTMENT has
as its value a *set of references* (that is, a set of OIDs) to objects of type EMPLOYEE;
these are the employees who work for the DEPARTMENT. The inverse is the refer-
ence attribute Dept of EMPLOYEE. We will see in Section 12.3 how the ODMG stan-
dard allows inverses to be explicitly declared as relationship attributes to ensure
that inverse references are consistent.

12.1.4 Encapsulation of Operations and Persistence of Objects

Encapsulation of Operations. The concept of *encapsulation* is one of the main
characteristics of OO languages and systems. It is also related to the concepts of
abstract data types and *information hiding* in programming languages. In tradi-
tional database models and systems this concept was not applied, since it is cus-
tomary to make the structure of database objects visible to users and external
programs. In these traditional models, a number of generic database operations

are applicable to objects *of all types*. For example, in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to *any relation* in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations. The concept of encapsulation is applied to database objects in ODBs by defining the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update. In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.

The external users of the object are only made aware of the **interface** of the operations, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of any hidden internal data structures of the object and the implementation of the operations that access these structures. The interface part of an operation is sometimes called the **signature**, and the operation implementation is sometimes called the **method**.

For database applications, the requirement that all objects be completely encapsulated is too stringent. One way to relax this requirement is to divide the structure of an object into **visible** and **hidden** attributes (instance variables). Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language. The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most ODMSs employ high-level query languages for accessing visible attributes. In Section 12.5 we will describe the OQL query language that is proposed as a standard query language for ODBs.

The term **class** is often used to refer to a type definition, along with the definitions of the operations for that type.[8] Figure 12.2 shows how the type definitions in Figure 12.1 can be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition. A method (implementation) for each operation must be defined elsewhere using a programming language. Typical operations include the **object constructor** operation (often called *new*), which is used to create a new object, and the **destructor** operation, which is used to destroy (delete) an object. A number of **object modifier** operations can also be declared to modify the states (values) of various attributes of an object. Additional operations can **retrieve** information about the object.

---

[8]This definition of *class* is similar to how it is used in the popular C++ programming language. The ODMG standard uses the word *interface* in addition to *class* (see Section 12.3). In the EER model, the term *class* was used to refer to an object type, along with the set of all objects of that type (see Chapter 8).

```
                  define class EMPLOYEE
                      type tuple  (   Fname:              string;
                                      Minit:              char;
                                      Lname:              string;
                                      Ssn:                string;
                                      Birth_date:         DATE;
                                      Address:            string;
                                      Sex:                char;
                                      Salary:             float;
                                      Supervisor:         EMPLOYEE;
                                      Dept:               DEPARTMENT; );
                      operations  age:                    integer;
                                  create_emp:             EMPLOYEE;
                                  destroy_emp:            boolean;
                  end EMPLOYEE;
                  define class DEPARTMENT
                      type tuple  (   Dname:              string;
                                      Dnumber:            integer;
                                      Mgr:                tuple ( Manager:      EMPLOYEE;
                                                                  Start_date:  DATE; );
                                      Locations:          set (string);
                                      Employees:          set (EMPLOYEE);
                                      Projects            set(PROJECT); );
                      operations  no_of_emps:             integer;
                                  create_dept:            DEPARTMENT;
                                  destroy_dept:           boolean;
                                  assign_emp(e: EMPLOYEE): boolean;
                                  (* adds an employee to the department *)
                                  remove_emp(e: EMPLOYEE): boolean;
                                  (* removes an employee from the department *)
                  end DEPARTMENT;
```

Figure 12.2
Adding operations to
the definitions of
EMPLOYEE and
DEPARTMENT.

An operation is typically applied to an object by using the **dot notation**. For example, if d is a reference to a DEPARTMENT object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted). The only exception is the constructor operation, which returns a reference to a new DEPARTMENT object. Hence, it is customary in some OO models to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure 12.2.[9] The dot notation is also used to refer to attributes of an object—for example, by writing d.Dnumber or d.Mgr_Start_date.

---

[9]Default names for the constructor and destructor operations exist in the C++ programming language. For example, for class EMPLOYEE, the *default constructor name* is EMPLOYEE and the *default destructor name* is ~EMPLOYEE. It is also common to use the *new* operation to create *new* objects.

Specifying Object Persistence via Naming and Reachability. An ODBS is often closely coupled with an object-oriented programming language (OOPL). The OOPL is used to specify the method (operation) implementations as well as other application code. Not all objects are meant to be stored permanently in the database. **Transient objects** exist in the executing program and disappear once the program terminates. **Persistent objects** are stored in the database and persist after program termination. The typical mechanisms for making an object persistent are *naming* and *reachability*.

The **naming mechanism** involves giving an object a unique persistent name within a particular database. This persistent **object name** can be given via a specific statement or operation in the program, as shown in Figure 12.3. The named persistent objects are used as **entry points** to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability**. The reachability mechanism works by making the object reachable from some other persistent object. An object *B* is said to be **reachable** from an object *A* if a sequence of references in the database lead from object *A* to object *B*.

If we first create a named persistent object *N*, whose state is a *set* of objects of some class *C*, we can make objects of *C* persistent by *adding them* to the set, thus making them reachable from *N*. Hence, *N* is a named object that defines a **persistent collection** of objects of class *C*. In the object model standard, *N* is called the **extent** of *C* (see Section 12.3).

For example, we can define a class DEPARTMENT_SET (see Figure 12.3) whose objects are of type set(DEPARTMENT).[10] We can create an object of type DEPARTMENT_SET, and give it a persistent name ALL_DEPARTMENTS, as shown in Figure 12.3. Any DEPARTMENT object that is added to the set of ALL_DEPARTMENTS by using the add_dept operation becomes persistent by virtue of its being reachable from ALL_DEPARTMENTS. As we will see in Section 12.3, the ODMG ODL standard gives the schema designer the option of naming an extent as part of class definition.

Notice the difference between traditional database models and ODBs in this respect. In traditional database models, such as the relational model, *all* objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the *type declaration* for EMPLOYEE and a *persistent set* of *all* EMPLOYEE records (tuples). In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) whose value is the *collection of references* (OIDs) to all persistent EMPLOYEE objects, if this is desired, as shown in Figure 12.3.[11] This allows transient and persistent objects to follow the

---

[10]As we will see in Section 12.3, the ODMG ODL syntax uses **set**<DEPARTMENT> instead of **set**(DEPARTMENT).

[11]Some systems, such as POET, automatically create the extent for a class.

```
        define class DEPARTMENT_SET
            type set (DEPARTMENT);
            operations add_dept(d: DEPARTMENT):  boolean;
                (* adds a department to the DEPARTMENT_SET object *)
                    remove_dept(d: DEPARTMENT):  boolean;
                (* removes a department from the DEPARTMENT_SET object *)
                    create_dept_set:       DEPARTMENT_SET;
                    destroy_dept_set:      boolean;
        end Department_Set;
        …
        persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
        (* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
        …
        d:= create_dept;
        (* create a new DEPARTMENT object in the variable d *)
        …
        b:= ALL_DEPARTMENTS.add_dept(d);
        (* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

Figure 12.3
Creating persistent
objects by naming
and reachability.

same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

12.1.5 Type Hierarchies and Inheritance

Simplified Model for Inheritance. Another main characteristic of ODBs is that they allow type hierarchies and inheritance. We use a simple OO model in this section—a model in which attributes and operations are treated uniformly—since both attributes and operations can be inherited. In Section 12.3, we will discuss the inheritance model of the ODMG standard, which differs from the model discussed here because it distinguishes between *two types of inheritance*. Inheritance allows the definition of new types based on other predefined types, leading to a **type** (or **class**) **hierarchy**.

A type is defined by assigning it a type name and then defining a number of attributes (instance variables) and operations (methods) for the type.[12] In the simplified model we use in this section, the attributes and operations are together called *functions,* since attributes resemble functions with zero arguments. A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method). We use the term **function** to refer to both attributes *and* operations, since they are treated similarly in a basic introduction to inheritance.[13]

---

[12]In this section we will use the terms *type* and *class* as meaning the same thing—namely, the attributes *and* operations of some type of object.

[13]We will see in Section 12.3 that types with functions are similar to the concept of interfaces as used in ODMG ODL.

A type in its simplest form has a **type name** and a list of visible (*public*) **functions**. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

TYPE_NAME: function, function, … , function

For example, a type that describes characteristics of a PERSON may be defined as follows:

PERSON: Name, Address, Birth_date, Age, Ssn

In the PERSON type, the Name, Address, Ssn, and Birth_date functions can be implemented as stored attributes, whereas the Age function can be implemented as an operation that calculates the Age from the value of the Birth_date attribute and the current date.

The concept of **subtype** is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which is referred to as the **supertype**. For example, suppose that we want to define two new types EMPLOYEE and STUDENT as follows:

EMPLOYEE: Name, Address, Birth_date, Age, Ssn, Salary, Hire_date, Seniority
STUDENT: Name, Address, Birth_date, Age, Ssn, Major, Gpa

Since both STUDENT and EMPLOYEE include all the functions defined for PERSON plus some additional functions of their own, we can declare them to be **subtypes** of PERSON. Each will inherit the previously defined functions of PERSON—namely, Name, Address, Birth_date, Age, and Ssn. For STUDENT, it is only necessary to define the new (local) functions Major and Gpa, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas Gpa may be implemented as an operation that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each STUDENT object as *hidden attributes*. For EMPLOYEE, the Salary and Hire_date functions may be stored attributes, whereas Seniority may be an operation that calculates Seniority from the value of Hire_date.

Therefore, we can declare EMPLOYEE and STUDENT as follows:

EMPLOYEE **subtype-of** PERSON: Salary, Hire_date, Seniority
STUDENT **subtype-of** PERSON: Major, Gpa

In general, a subtype includes *all* of the functions that are defined for its supertype plus some additional functions that are *specific* only to the subtype. Hence, it is possible to generate a **type hierarchy** to show the supertype/subtype relationships among all the types declared in the system.

As another example, consider a type that describes objects in plane geometry, which may be defined as follows:

GEOMETRY_OBJECT: Shape, Area, Reference_point

For the GEOMETRY_OBJECT type, Shape is implemented as an attribute (its domain can be an enumerated type with values 'triangle', 'rectangle', 'circle', and so on), and

Area is a method that is applied to calculate the area. Reference_point specifies the coordinates of a point that determines the object location. Now suppose that we want to define a number of subtypes for the GEOMETRY_OBJECT type, as follows:

> RECTANGLE **subtype-of** GEOMETRY_OBJECT: Width, Height
> TRIANGLE S **subtype-of** GEOMETRY_OBJECT: Side1, Side2, Angle
> CIRCLE **subtype-of** GEOMETRY_OBJECT: Radius

Notice that the Area operation may be implemented by a different method for each subtype, since the procedure for area calculation is different for rectangles, triangles, and circles. Similarly, the attribute Reference_point may have a different meaning for each subtype; it might be the center point for RECTANGLE and CIRCLE objects, and the vertex point between the two given sides for a TRIANGLE object.

Notice that type definitions describe objects but *do not* generate objects on their own. When an object is created, typically it belongs to one or more of these types that have been declared. For example, a circle object is of type CIRCLE and GEOMETRY_OBJECT (by inheritance). Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are persistently stored in the database.

Constraints on Extents Corresponding to a Type Hierarchy. In most ODBs, an **extent** is defined to store the collection of persistent objects for each type or subtype. In this case, the constraint is that every object in an extent that corresponds to a subtype must also be a member of the *extent* that corresponds to its supertype. Some OO database systems have a predefined system type (called the ROOT class or the OBJECT class) whose extent contains all the objects in the system.[14]

Classification then proceeds by assigning objects into additional subtypes that are meaningful to the application, creating a **type hierarchy** (or **class hierarchy**) for the system. All extents for system- and user-defined classes are subsets of the extent corresponding to the class OBJECT, directly or indirectly. In the ODMG model (see Section 12.3), the user may or may not specify an extent for each class (type), depending on the application.

An extent is a named persistent object whose value is a **persistent collection** that holds a collection of objects of the same type that are stored permanently in the database. The objects can be accessed and shared by multiple programs. It is also possible to create a **transient collection**, which exists temporarily during the execution of a program but is not kept when the program terminates. For example, a transient collection may be created in a program to hold the result of a query that selects some objects from a persistent collection and copies those objects into the transient collection. The program can then manipulate the objects in the transient collection, and once the program terminates, the transient collection ceases to exist. In general, numerous collections—transient or persistent—may contain objects of the same type.

---

[14]This is called OBJECT in the ODMG model (see Section 12.3).

The inheritance model discussed in this section is very simple. As we will see in Section 12.3, the ODMG model distinguishes between type inheritance—called *interface inheritance* and denoted by a colon (:)—and the *extent inheritance* constraint—denoted by the keyword EXTEND.

12.1.6 Other Object-Oriented Concepts

Polymorphism of Operations (Operator Overloading). Another characteristic of OO systems in general is that they provide for **polymorphism** of operations, which is also known as **operator overloading**. This concept allows the same *operator name* or *symbol* to be bound to two or more different *implementations* of the operator, depending on the type of objects to which the operator is applied. A simple example from programming languages can illustrate this concept. In some languages, the operator symbol "+" can mean different things when applied to operands (objects) of different types. If the operands of "+" are of type *integer*, the operation invoked is integer addition. If the operands of "+" are of type *floating point*, the operation invoked is floating-point addition. If the operands of "+" are of type *set*, the operation invoked is set union. The compiler can determine which operation to execute based on the types of operands supplied.

In OO databases, a similar situation may occur. We can use the GEOMETRY_OBJECT example presented in Section 12.1.5 to illustrate operation polymorphism[15] in ODB. In this example, the function Area is declared for all objects of type GEOMETRY_OBJECT. However, the implementation of the method for Area may differ for each subtype of GEOMETRY_OBJECT. One possibility is to have a general implementation for calculating the area of a generalized GEOMETRY_OBJECT (for example, by writing a general algorithm to calculate the area of a polygon) and then to rewrite more efficient algorithms to calculate the areas of specific types of geometric objects, such as a circle, a rectangle, a triangle, and so on. In this case, the Area function is *overloaded* by different implementations.

The ODMS must now select the appropriate method for the Area function based on the type of geometric object to which it is applied. In strongly typed systems, this can be done at compile time, since the object types must be known. This is termed **early** (or **static**) **binding**. However, in systems with weak typing or no typing (such as Smalltalk, LISP, PHP, and most scripting languages), the type of the object to which a function is applied may not be known until runtime. In this case, the function must check the type of object at runtime and then invoke the appropriate method. This is often referred to as **late** (or **dynamic**) **binding**.

Multiple Inheritance and Selective Inheritance. **Multiple inheritance** occurs when a certain subtype *T* is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes. For example, we may create

---

[15]In programming languages, there are several kinds of polymorphism. The interested reader is referred to the Selected Bibliography at the end of this chapter for works that include a more thorough discussion.

a subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER. This leads to the creation of a **type lattice** rather than a type hierarchy. One problem that can occur with multiple inheritance is that the supertypes from which the subtype inherits may have distinct functions of the same name, creating an ambiguity. For example, both MANAGER and ENGINEER may have a function called Salary. If the Salary function is implemented by different methods in the MANAGER and ENGINEER supertypes, an ambiguity exists as to which of the two is inherited by the subtype ENGINEERING_MANAGER. It is possible, however, that both ENGINEER and MANAGER inherit Salary from the same supertype (such as EMPLOYEE) higher up in the lattice. The general rule is that if a function is inherited from some *common supertype,* then it is inherited only once. In such a case, there is no ambiguity; the problem only arises if the functions are distinct in the two supertypes.

There are several techniques for dealing with ambiguity in multiple inheritance. One solution is to have the system check for ambiguity when the subtype is created, and to let the user explicitly choose which function is to be inherited at this time. A second solution is to use some system default. A third solution is to disallow multiple inheritance altogether if name ambiguity occurs, instead forcing the user to change the name of one of the functions in one of the supertypes. Indeed, some OO systems do not permit multiple inheritance at all. In the object database standard (see Section 12.3), multiple inheritance is allowed for operation inheritance of interfaces, but is not allowed for EXTENDS inheritance of classes.

Selective inheritance occurs when a subtype inherits only some of the functions of a supertype. Other functions are not inherited. In this case, an EXCEPT clause may be used to list the functions in a supertype that are *not* to be inherited by the subtype. The mechanism of selective inheritance is not typically provided in ODBs, but it is used more frequently in artificial intelligence applications.[16]

12.1.7 Summary of Object Database Concepts

To conclude this section, we give a summary of the main concepts used in ODBs and object-relational systems:

- **Object identity.** Objects have unique identities that are independent of their attribute values and are generated by the ODB system.
- **Type constructors.** Complex object structures can be constructed by applying in a nested manner a set of basic type generators/constructors, such as tuple, set, list, array, and bag.
- **Encapsulation of operations.** Both the object structure and the operations that can be applied to individual objects are included in the class/type definitions.
- **Programming language compatibility.** Both persistent and transient objects are handled seamlessly. Objects are made persistent by being reachable from

---

[16]In the ODMG model, type inheritance refers to inheritance of operations only, not attributes (see Section 12.3).

a persistent collection (extent) or by explicit naming (assigning a unique name by which the object can be referenced/retrieved).

- **Type hierarchies and inheritance.** Object types can be specified by using a type hierarchy, which allows the inheritance of both attributes and methods (operations) of previously defined types. Multiple inheritance is allowed in some models.

- **Extents.** All persistent objects of a particular class/type C can be stored in an extent, which is a named persistent object of type set(C). Extents corresponding to a type hierarchy have set/subset constraints enforced on their collections of persistent objects.

- **Polymorphism and operator overloading.** Operations and method names can be overloaded to apply to different object types with different implementations.

In the following sections we show how these concepts are realized, first in the SQL standard (Section 12.2) and then in the ODMG standard (Section 12.3).

12.2 Object Database Extensions to SQL

We introduced SQL as the standard language for RDBMSs in Chapters 6 and 7. As we discussed, SQL was first specified by Chamberlin and Boyce (1974) and underwent enhancements and standardization in 1989 and 1992. The language continued its evolution with a new standard, initially called SQL3 while being developed and later known as SQL:99 for the parts of SQL3 that were approved into the standard. Starting with the version of SQL known as SQL3, features from object databases were incorporated into the SQL standard. At first, these extensions were known as SQL/Object, but later they were incorporated in the main part of SQL, known as SQL/Foundation in SQL:2008.

The relational model with object database enhancements is sometimes referred to as the **object-relational model**. Additional revisions were made to SQL in 2003 and 2006 to add features related to XML (see Chapter 13).

The following are some of the object database features that have been included in SQL:

- Some **type constructors** have been added to specify complex objects. These include the *row type*, which corresponds to the tuple (or struct) constructor. An *array type* for specifying collections is also provided. Other collection type constructors, such as *set*, *list*, and *bag* constructors, were not part of the original SQL/Object specifications in SQL:99 but were later included in the standard in SQL:2008.

- A mechanism for specifying **object identity** through the use of *reference type* is included.

- **Encapsulation of operations** is provided through the mechanism of user-defined types (UDTs) that may include operations as part of their declaration. These are somewhat similar to the concept of *abstract data*

types that were developed in programming languages. In addition, the concept of user-defined routines (UDRs) allows the definition of general methods (operations).

■ **Inheritance** mechanisms are provided using the keyword UNDER.

We now discuss each of these concepts in more detail. In our discussion, we will refer to the example in Figure 12.4.

12.2.1 User-Defined Types Using CREATE TYPE and Complex Objects

To allow the creation of complex-structured objects and to separate the declaration of a class/type from the creation of a table (which is the collection of objects/rows and hence corresponds to the extent discussed in Section 12.1), SQL now provides **user-defined types** (**UDT**s). In addition, four collection types have been included to allow for collections (multivalued types and attributes) in order to specify complex-structured objects rather than just simple (flat) records. The user will create the UDTs for a particular application as part of the database schema. A **UDT** may be specified in its simplest form using the following syntax:

    CREATE TYPE TYPE_NAME **AS** (<component declarations>);

Figure 12.4 illustrates some of the object concepts in SQL. We will explain the examples in this figure gradually as we explain the concepts. First, a UDT can be used as either the type for an attribute or as the type for a table. By using a UDT as the type for an attribute within another UDT, a complex structure for objects (tuples) in a table can be created, much like that achieved by nesting type constructors/generators as discussed in Section 12.1. This is similar to using the *struct* type constructor of Section 12.1.3. For example, in Figure 12.4(a), the UDT STREET_ADDR_TYPE is used as the type for the STREET_ADDR attribute in the UDT USA_ADDR_TYPE. Similarly, the UDT USA_ADDR_TYPE is in turn used as the type for the ADDR attribute in the UDT PERSON_TYPE in Figure 12.4(b). If a UDT does not have any operations, as in the examples in Figure 12.4(a), it is possible to use the concept of **ROW TYPE** to directly create a structured attribute by using the keyword **ROW**. For example, we could use the following instead of declaring STREET_ADDR_TYPE as a separate type as in Figure 12.4(a):

```
CREATE TYPE USA_ADDR_TYPE AS (
    STREET_ADDR    ROW (  NUMBER          VARCHAR (5),
                          STREET_NAME     VARCHAR (25),
                          APT_NO          VARCHAR (5),
                          SUITE_NO        VARCHAR (5) ),
    CITY           VARCHAR (25),
    ZIP            VARCHAR (10)
);
```

To allow for collection types in order to create complex-structured objects, four constructors are now included in SQL: ARRAY, MULTISET, LIST, and SET. These are

(a) CREATE TYPE STREET_ADDR_TYPE **AS** (
    NUMBER   VARCHAR (5),
    STREET   NAME VARCHAR (25),
    APT_NO   VARCHAR (5),
    SUITE_NO   VARCHAR (5)
);
CREATE TYPE USA_ADDR_TYPE **AS** (
    STREET_ADDR   STREET_ADDR_TYPE,
    CITY   VARCHAR (25),
    ZIP   VARCHAR (10)
);
CREATE TYPE USA_PHONE_TYPE **AS** (
    PHONE_TYPE   VARCHAR (5),
    AREA_CODE   CHAR (3),
    PHONE_NUM   CHAR (7)
);

(b) CREATE TYPE PERSON_TYPE **AS** (
    NAME   VARCHAR (35),
    SEX   CHAR,
    BIRTH_DATE   DATE,
    PHONES   USA_PHONE_TYPE ARRAY [4],
    ADDR   USA_ADDR_TYPE
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD AGE() **RETURNS INTEGER**;
CREATE INSTANCE METHOD AGE() **RETURNS INTEGER**
    FOR PERSON_TYPE
    BEGIN
        RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                TODAY'S DATE AND SELF.BIRTH_DATE */
    END;
);

(c) CREATE TYPE GRADE_TYPE **AS** (
    COURSENO   CHAR (8),
    SEMESTER   VARCHAR (8),
    YEAR   CHAR (4),
    GRADE   CHAR
);
CREATE TYPE STUDENT_TYPE **UNDER** PERSON_TYPE **AS** (
    MAJOR_CODE   CHAR (4),
    STUDENT_ID   CHAR (12),
    DEGREE   VARCHAR (5),
    TRANSCRIPT   GRADE_TYPE ARRAY [100]   (continues)

Figure 12.4
Illustrating some of the object features of SQL. (a) Using UDTs as types for attributes such as Address and Phone, (b) specifying UDT for PERSON_TYPE, (c) specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE.

Figure 12.4 (continued)
llustrating some of the object features of SQL. (c) (continued) Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE, (d) Creating tables based on some of the UDTs, and illustrating table inheritance, (e) Specifying relationships using REF and SCOPE.

```
INSTANTIABLE
NOT FINAL
INSTANCE METHOD GPA( ) RETURNS FLOAT;
CREATE INSTANCE METHOD GPA( ) RETURNS FLOAT
    FOR STUDENT_TYPE
    BEGIN
        RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                    SELF.TRANSCRIPT */
    END;
);
CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
    JOB_CODE        CHAR (4),
    SALARY          FLOAT,
    SSN             CHAR (11)
INSTANTIABLE
NOT FINAL
);
CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
    DEPT_MANAGED CHAR (20)
INSTANTIABLE
);
```

```
(d) CREATE TABLE PERSON OF PERSON_TYPE
        REF IS PERSON_ID SYSTEM GENERATED;
    CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
        UNDER PERSON;
    CREATE TABLE MANAGER OF MANAGER_TYPE
        UNDER EMPLOYEE;
    CREATE TABLE STUDENT OF STUDENT_TYPE
        UNDER PERSON;
```

```
(e) CREATE TYPE COMPANY_TYPE AS (
        COMP_NAME     VARCHAR (20),
        LOCATION      VARCHAR (20));
    CREATE TYPE EMPLOYMENT_TYPE AS (
        Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
        Company REF (COMPANY_TYPE) SCOPE (COMPANY) );
    CREATE TABLE COMPANY OF COMPANY_TYPE (
        REF IS COMP_ID SYSTEM GENERATED,
        PRIMARY KEY (COMP_NAME) );
    CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
```

similar to the type constructors discussed in Section 12.1.3. In the initial specification of SQL/Object, only the ARRAY type was specified, since it can be used to simulate the other types, but the three additional collection types were included in a later version of the SQL standard. In Figure 12.4(b), the PHONES attribute of PERSON_TYPE has as its type an array whose elements are of the previously defined UDT USA_PHONE_TYPE. This array has a maximum of four elements, meaning that we can store up to four phone numbers per person. An array can also have no maximum number of elements if desired.

An array type can have its elements referenced using the common notation of square brackets. For example, PHONES[1] refers to the first location value in a PHONES attribute (see Figure 12.4(b)). A built-in function **CARDINALITY** can return the current number of elements in an array (or any other collection type). For example, PHONES[**CARDINALITY** (PHONES)] refers to the last element in the array.

The commonly used dot notation is used to refer to components of a **ROW TYPE** or a UDT. For example, ADDR.CITY refers to the CITY component of an ADDR attribute (see Figure 12.4(b)).

12.2.2 Object Identifiers Using Reference Types

Unique system-generated object identifiers can be created via the **reference type** using the keyword **REF**. For example, in Figure 12.4(b), the phrase:

REF IS SYSTEM GENERATED

indicates that whenever a new PERSON_TYPE object is created, the system will assign it a unique system-generated identifier. It is also possible not to have a system-generated object identifier and use the traditional keys of the basic relational model if desired.

In general, the user can specify that system-generated object identifiers for the individual rows in a table should be created. By using the syntax:

REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;

the user declares that the attribute named <OID_ATTRIBUTE> will be used to identify individual tuples in the table. The options for <VALUE_GENERATION_METHOD> are SYSTEM GENERATED or DERIVED. In the former case, the system will automatically generate a unique identifier for each tuple. In the latter case, the traditional method of using the user-provided primary key value to identify tuples is applied.

12.2.3 Creating Tables Based on the UDTs

For each UDT that is specified to be instantiable via the phrase **INSTANTIABLE** (see Figure 12.4(b)), one or more tables may be created. This is illustrated in Figure 12.4(d), where we create a table PERSON based on the PERSON_TYPE UDT. Notice that the UDTs in Figure 12.4(a) are *noninstantiable* and hence can only be used as

types for attributes, but not as a basis for table creation. In Figure 12.4(b), the attribute PERSON_ID will hold the system-generated object identifier whenever a new PERSON record (object) is created and inserted in the table.

12.2.4 Encapsulation of Operations

In SQL, a **user-defined type** can have its own behavioral specification by specifying methods (or operations) in addition to the attributes. The general form of a UDT specification with methods is as follows:

CREATE TYPE <TYPE-NAME> (
    <LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>
    <DECLARATION OF FUNCTIONS (METHODS)>
);

For example, in Figure 12.4(b), we declared a method Age() that calculates the age of an individual object of type PERSON_TYPE.

The code for implementing the method still has to be written. We can refer to the method implementation by specifying the file that contains the code for the method, or we can write the actual code within the type declaration itself (see Figure 12.4(b)).

SQL provides certain built-in functions for user-defined types. For a UDT called TYPE_T, the **constructor function** TYPE_T( ) returns a new object of that type. In the new UDT object, every attribute is initialized to its default value. An **observer function** A is implicitly created for each attribute A to read its value. Hence, $A(X)$ or $X.A$ returns the value of attribute A of TYPE_T if X is a variable that refers to an object/row of type TYPE_T. A **mutator function** for updating an attribute sets the value of the attribute to a new value. SQL allows these functions to be blocked from public use; an EXECUTE privilege is needed to have access to these functions.

In general, a UDT can have a number of user-defined functions associated with it. The syntax is

INSTANCE METHOD <NAME> (<ARGUMENT_LIST>) **RETURNS**
<RETURN_TYPE>;

Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM language of SQL (see Chapter 10). External functions are written in a host language, with only their signature (interface) appearing in the UDT definition. An external function definition can be declared as follows:

DECLARE EXTERNAL <FUNCTION_NAME> <SIGNATURE>
LANGUAGE <LANGUAGE_NAME>;

Attributes and functions in UDTs are divided into three categories:

- PUBLIC (visible at the UDT interface)
- PRIVATE (not visible at the UDT interface)
- PROTECTED (visible only to subtypes)

It is also possible to define virtual attributes as part of UDTs, which are computed and updated using functions.

12.2.5 Specifying Inheritance and Overloading of Functions

In SQL, inheritance can be applied to types or to tables; we will discuss the meaning of each in this section. Recall that we already discussed many of the principles of inheritance in Section 12.1.5. SQL has rules for dealing with **type inheritance** (specified via the **UNDER** keyword). In general, both attributes and instance methods (operations) are inherited. The phrase **NOT FINAL** must be included in a UDT if subtypes are allowed to be created under that UDT (see Figures 12.4(a) and (b), where PERSON_TYPE, STUDENT_TYPE, and EMPLOYEE_TYPE are declared to be NOT FINAL). Associated with type inheritance are the rules for overloading of function implementations and for resolution of function names. These inheritance rules can be summarized as follows:

- All attributes are inherited.
- The order of supertypes in the UNDER clause determines the inheritance hierarchy.
- An instance of a subtype can be used in every context in which a supertype instance is used.
- A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
- When a function is called, the best match is selected based on the types of all arguments.
- For dynamic linking, the types of the parameters are considered at runtime.

Consider the following examples to illustrate type inheritance, which are illustrated in Figure 12.4(c). Suppose that we want to create two subtypes of PERSON_TYPE: EMPLOYEE_TYPE and STUDENT_TYPE. In addition, we also create a subtype MANAGER_TYPE that inherits all the attributes (and methods) of EMPLOYEE_TYPE but has an additional attribute DEPT_MANAGED. These subtypes are shown in Figure 12.4(c).

In general, we specify the local (specific) attributes and any additional specific methods for the subtype, which inherits the attributes and operations (methods) of its supertype.

Another facility in SQL is **table inheritance** via the supertable/subtable facility. This is also specified using the keyword **UNDER** (see Figure 12.4(d)). Here, a new record that is inserted into a subtable, say the MANAGER table, is also inserted into its supertables EMPLOYEE and PERSON. Notice that when a record is inserted in MANAGER, we must provide values for all its inherited attributes. INSERT, DELETE, and UPDATE operations are appropriately propagated. Basically, table inheritance corresponds to the *extent inheritance* discussed in Section 12.1.5. The rule is that a tuple in a sub-table must also exist in its super-table to enforce the set/subset constraint on the objects.

12.2.6 Specifying Relationships via Reference

A component attribute of one tuple may be a **reference** (specified using the keyword **REF**) to a tuple of another (or possibly the same) table. An example is shown in Figure 12.4(e).

The keyword **SCOPE** specifies the name of the table whose tuples can be referenced by the reference attribute. Notice that this is similar to a foreign key, except that the system-generated OID value is used rather than the primary key value.

SQL uses a **dot notation** to build **path expressions** that refer to the component attributes of tuples and row types. However, for an attribute whose type is REF, the dereferencing symbol –> is used. For example, the query below retrieves employees working in the company named 'ABCXYZ' by querying the EMPLOYMENT table:

```
SELECT    E.Employee–>NAME
FROM      EMPLOYMENT AS E
WHERE     E.Company–>COMP_NAME = 'ABCXYZ';
```

In SQL, –> is used for **dereferencing** and has the same meaning assigned to it in the C programming language. Thus, if *r* is a reference to a tuple (object) and *a* is a component attribute in that tuple, then *r* –> *a* is the value of attribute *a* in that tuple.

If several relations of the same type exist, SQL provides the SCOPE keyword by which a reference attribute may be made to point to a tuple within a specific table of that type.

12.3 The ODMG Object Model and the Object Definition Language ODL

As we discussed in the introduction to Chapter 6, one of the reasons for the success of commercial relational DBMSs is the SQL standard. The lack of a standard for ODBs for several years may have caused some potential users to shy away from converting to this new technology. Subsequently, a consortium of ODB vendors and users, called ODMG (Object Data Management Group), proposed a standard that is known as the ODMG-93 or ODMG 1.0 standard. This was revised into ODMG 2.0, and later to ODMG 3.0. The standard is made up of several parts, including the **object model**, the **object definition language** (**ODL**), the **object query language** (**OQL**), and the **bindings** to object-oriented programming languages.

In this section, we describe the ODMG object model and the ODL. In Section 12.4, we discuss how to design an ODB from an EER conceptual schema. We will give an overview of OQL in Section 12.5, and the C++ language binding in Section 12.6. Examples of how to use ODL, OQL, and the C++ language binding will use the UNIVERSITY database example introduced in Chapter 4. In our description, we will follow the ODMG 3.0 object model as described in Cattell et al. (2000).[17] It is

---

[17]The earlier versions of the object model were published in 1993 and 1997.

important to note that many of the ideas embodied in the ODMG object model are based on two decades of research into conceptual modeling and object databases by many researchers.

The incorporation of object concepts into the SQL relational database standard, leading to object-relational technology, was presented in Section 12.2.

12.3.1 Overview of the Object Model of ODMG

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts. We will try to adhere to the ODMG terminology in this chapter. Many of the concepts in the ODMG model have already been discussed in Section 12.1, and we assume the reader has read this section. We will point out whenever the ODMG terminology differs from that used in Section 12.1.

Objects and Literals. Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a **state** (or current value), whereas a literal has a value (state) but *no object identifier.*[18] In either case, the value can have a complex structure. The object state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.

An **object** has five aspects: identifier, name, lifetime, structure, and creation.

1. The **object identifier** is a unique system-wide identifier (or **Object_id**).[19] Every object must have an object identifier.

2. Some objects may optionally be given a unique **name** within a particular ODMS—this name can be used to locate the object, and the system should return the object given that name.[20] Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object class/type—such as *extents*—will have a name. These names are used as **entry points** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names, and it is possible to give *more than one* name to an object. All names within a particular ODB must be unique.

---

[18]We will use the terms *value* and *state* interchangeably here.

[19]This corresponds to the OID of Section 12.1.2.

[20]This corresponds to the naming mechanism for persistence, described in Section 12.1.4.

3. The **lifetime** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing program that disappears after the program terminates). Lifetimes are independent of classes/types—that is, some objects of a particular class may be transient whereas others may be persistent.

4. The **structure** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is *atomic* or not. An **atomic object** refers to a single object that follows a user-defined type, such as Employee or Department. If an object is not atomic, then it will be composed of other objects. For example, a *collection object* is not an atomic object, since its state will be a collection of other objects.[21] The term *atomic object* is different from how we defined the *atom constructor* in Section 12.1.3, which referred to all values of built-in data types. In the ODMG model, an atomic object is any *individual user-defined object*. All values of the basic built-in data types are considered to be *literals*.

5. Object **creation** refers to the manner in which an object can be created. This is typically accomplished via an operation *new* for a special Object_Factory interface. We shall describe this in more detail later in this section.

In the object model, a **literal** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure. There are three types of literals: atomic, structured, and collection.

1. **Atomic literals**[22] correspond to the values of basic data types and are predefined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords **long**, **short**, **unsigned long**, and **unsigned short** in ODL), regular and double precision floating-point numbers (**float, double**), Boolean values (**boolean**), single characters (**char**), character strings (**string**), and enumeration types (**enum**), among others.

2. **Structured literals** correspond roughly to values that are constructed using the tuple constructor described in Section 12.1.3. The built-in structured literals include Date, Interval, Time, and Timestamp (see Figure 12.5(b)). Additional user-defined structured literals can be defined as needed by each application.[23] User-defined structures are created using the **STRUCT** keyword in ODL, as in the C and C++ programming languages.

---

[21]In the ODMG model, *atomic objects* do not correspond to objects whose values are basic data types. All basic values (integers, reals, and so on) are considered *literals*.

[22]The use of the word *atomic* in *atomic literal* corresponds to the way we used atom constructor in Section 12.1.3.

[23]The structures for Date, Interval, Time, and Timestamp can be used to create either literal values or objects with identifiers.

(a) nterface Object {

 …

  boolean   same_as(in object other_object);

  object   copy();

  void   delete();

 };

(b) Class Date : Object {

  enum   Weekday

          { Sunday, Monday, Tuesday, Wednesday,

           Thursday, Friday, Saturday };

  enum   Month

          { January, February, March, April, May, June,

           July, August, September, October, November,

           December };

  unsigned short   year();

  unsigned short   month();

  unsigned short   day();

  …

  boolean   is_equal(in Date other_date);

  boolean   is_greater(in Date other_date);

  … };

 Class Time : Object {

  …

  unsigned short   hour();

  unsigned short   minute();

  unsigned short   second();

  unsigned short   millisecond();

  …

  boolean   is_equal(in Time a_time);

  boolean   is_greater(in Time a_time);

  …

  Time   add_interval(in Interval an_interval);

  Time   subtract_interval(in Interval an_interval);

  Interval   subtract_time(in Time other_time); };

 class Timestamp : Object {

  …

  unsigned short   year();

  unsigned short   month();

  unsigned short   day();

  unsigned short   hour();

  unsigned short   minute();

  unsigned short   second();

  unsigned short   millisecond();

  …

  Timestamp   plus(in Interval an_interval);

Figure 12.5
Overview of the interface definitions for part of the ODMG object model. (a) The basic Object interface, inherited by all objects, (b) Some standard interfaces for structured literals.

(continues)

**Figure 12.5
(continued)**
Overview of the
interface
definitions for
part of the ODMG
object model.
(b) (continued) Some
standard interfaces
for structured literals,
(c) Interfaces for
collections and
iterators.

```
Timestamp            minus(in Interval an_interval);
boolean              is_equal(in Timestamp a_timestamp);
boolean              is_greater(in Timestamp a_timestamp);
… };
class Interval :     Object {
unsigned short       day();
unsigned short       hour();
unsigned short       minute();
unsigned short       second();
unsigned short       millisecond();
…
Interval             plus(in Interval an_interval);
Interval             minus(in Interval an_interval);
Interval             product(in long a_value);
Interval             quotient(in long a_value);
boolean              is_equal(in interval an_interval);
boolean              is_greater(in interval an_interval);
… };
```

(c)   interface Collection : Object {

```
…
exception            ElementNotFound{ Object element; };
unsigned long        cardinality();
boolean              is_empty();
…
boolean              contains_element(in Object element);
void                 insert_element(in Object element);
void                 remove_element(in Object element)
                           raises(ElementNotFound);
iterator             create_iterator(in boolean stable);
… };
interface Iterator {
exception            NoMoreElements();
…
boolean              at_end();
void                 reset();
Object               get_element() raises(NoMoreElements);
void                 next_position() raises(NoMoreElements);
… };
interface set : Collection {
set                  create_union(in set other_set);
…
boolean              is_subset_of(in set other_set);
… };
interface bag : Collection {
unsigned long        occurrences_of(in Object element);
```

```
    bag                 create_union(in Bag other_bag);
    … };
interface list : Collection {
    exception           Invalid_Index{unsigned_long index; );
    void                remove_element_at(in unsigned long index)
                            raises(InvalidIndex);
    Object              retrieve_element_at(in unsigned long index)
                            raises(InvalidIndex);
    void                replace_element_at(in Object element, in unsigned long index)
                            raises(InvalidIndex);
    void                insert_element_after(in Object element, in unsigned long index)
                            raises(InvalidIndex);
    …
    void                insert_element_first(in Object element);
    …
    void                remove_first_element() raises(ElementNotFound);
    …
    Object              retrieve_first_element() raises(ElementNotFound);
    …
    list                concat(in list other_list);
    void                append(in list other_list);
};
interface array : Collection {
    exception           Invalid_Index{unsigned_long index; };
    exception           Invalid_Size{unsigned_long size; };
    void                remove_element_at(in unsigned long index)
                            raises(InvalidIndex);
    Object              retrieve_element_at(in unsigned long index)
                            raises(InvalidIndex);
    void                replace_element_at(in unsigned long index, in Object element)
                            raises(InvalidIndex);
    void                resize(in unsigned long new_size)
                            raises(InvalidSize);
};
struct association { Object key; Object value; };
interface dictionary : Collection {
    exception           DuplicateName{string key; };
    exception           KeyNotFound{Object key; };
    void                bind(in Object key, in Object value)
                            raises(DuplicateName);
    void                unbind(in Object key) raises(KeyNotFound);
    Object              lookup(in Object key) raises(KeyNotFound);
    boolean             contains_key(in Object key);
};
```

Figure 12.5 (continued)
Overview of the interface definitions for part of the ODMG object model.
(c) (continued) Interfaces for collections and iterators.

3. **Collection literals** specify a literal value that is a collection of objects or values but the collection itself does not have an Object_id. The collections in the object model can be defined by the *type generators* **set**$<T>$, **bag**$<T>$, **list**$<T>$, and **array**$<T>$, where T is the type of objects or values in the collection.[24] Another collection type is **dictionary**$<K, V>$, which is a collection of associations $<K, V>$, where each K is a key (a unique search value) associated with a value V; this can be used to create an index on a collection of values V.

Figure 12.5 gives a simplified view of the basic types and type generators of the object model. The notation of ODMG uses three concepts: interface, literal, and class. Following the ODMG terminology, we use the word **behavior** to refer to *operations* and **state** to refer to *properties* (attributes and relationships). An **interface** specifies only behavior of an object type and is typically **noninstantiable** (that is, no objects are created corresponding to an interface). Although an interface may have state properties (attributes and relationships) as part of its specifications, these *cannot* be inherited from the interface. Hence, an interface serves to define operations that can be *inherited* by other interfaces, as well as by classes that define the user-defined objects for a particular application. A **class** specifies both state (attributes) and behavior (operations) of an object type and is **instantiable**. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema. Finally, a **literal** declaration specifies state but no behavior. Thus, a literal instance holds a simple or complex structured value but has neither an object identifier nor encapsulated operations.

Figure 12.5 is a simplified version of the object model. For the full specifications, see Cattell et al. (2000). We will describe some of the constructs shown in Figure 12.5 as we describe the object model. In the object model, all objects inherit the basic interface operations of Object, shown in Figure 12.5(a); these include operations such as copy (creates a new copy of the object), delete (deletes the object), and same_as (compares the object's identity to another object).[25] In general, operations are applied to objects using the **dot notation**. For example, given an object O, to compare it with another object P, we write

O.same_as(P)

The result returned by this operation is Boolean and would be true if the identity of P is the same as that of O, and false otherwise. Similarly, to create a copy P of object O, we write

$P = O$.copy()

An alternative to the dot notation is the **arrow notation:** O–>same_as(P) or O–>copy().

---

[24]These are similar to the corresponding type constructors described in Section 12.1.3.

[25]Additional operations are defined on objects for *locking* purposes, which are not shown in Figure 12.5. We discuss locking concepts for databases in Chapter 22.

12.3.2 Inheritance in the Object Model of ODMG

In the ODMG object model, two types of inheritance relationships exist: behavior-only inheritance and state plus behavior inheritance. **Behavior inheritance** is also known as *ISA* or *interface inheritance* and is specified by the colon (:) notation.[26] Hence, in the ODMG object model, behavior inheritance requires the supertype to be an interface, whereas the subtype could be either a class or another interface.

The other inheritance relationship, called EXTENDS **inheritance,** is specified by the keyword extends. It is used to inherit both state and behavior strictly among classes, so both the supertype and the subtype must be classes. Multiple inheritance via extends is not permitted. However, multiple inheritance is allowed for behavior inheritance via the colon (:) notation. Hence, an interface may inherit behavior from several other interfaces. A class may also inherit behavior from several interfaces via colon (:) notation, in addition to inheriting behavior and state from *at most one* other class via extends. In Section 12.3.4 we will give examples of how these two inheritance relationships—":" and extends—may be used.

12.3.3 Built-in Interfaces and Classes in the Object Model

Figure 12.5 shows the built-in interfaces of the object model. All interfaces, such as Collection, Date, and Time, inherit the basic Object interface. In the object model, there is a distinction between collections, whose state contains multiple objects or literals, versus atomic (and structured) objects, whose state is an individual object or literal. **Collection objects** inherit the basic Collection interface shown in Figure 12.5(c), which shows the operations for all collection objects. Given a collection object O, the O.cardinality() operation returns the number of elements in the collection. The operation O.is_empty() returns true if the collection O is empty, and returns false otherwise. The operations O.insert_element(E) and O.remove_element(E) insert or remove an element E from the collection O. Finally, the operation O.contains_element(E) returns true if the collection O includes element E, and returns false otherwise. The operation $I = O$.create_iterator() creates an **iterator object** I for the collection object O, which can iterate over each element in the collection. The interface for iterator objects is also shown in Figure 12.5(c). The I.reset() operation sets the iterator at the first element in a collection (for an unordered collection, this would be some arbitrary element), and I.next_position() sets the iterator to the next element. The I.get_element() retrieves the **current element**, which is the element at which the iterator is currently positioned.

The ODMG object model uses **exceptions** for reporting errors or particular conditions. For example, the ElementNotFound exception in the Collection interface would be raised by the O.remove_element(E) operation if E is not an element in the collection O.

---

[26]The ODMG report also calls interface inheritance as type/subtype, is-a, and generalization/specialization relationships, although in the literature these terms have been used to describe inheritance of both state and operations (see Chapter 8 and Section 12.1).

The NoMoreElements exception in the iterator interface would be raised by the *I*.next_position() operation if the iterator is currently positioned at the last element in the collection, and hence no more elements exist for the iterator to point to.

Collection objects are further specialized into set, list, bag, array, and dictionary, which inherit the operations of the Collection interface. A **set<*T*> type generator** can be used to create objects such that the value of object *O* is a *set whose elements are of type T*. The Set interface includes the additional operation *P* = *O*.create_union(*S*) (see Figure 12.5(c)), which returns a new object *P* of type set<*T*> that is the union of the two sets *O* and *S*. Other operations similar to create_union (not shown in Figure 12.5(c)) are create_intersection(*S*) and create_difference(*S*). Operations for set comparison include the *O*.is_subset_of(*S*) operation, which returns true if the set object *O* is a subset of some other set object *S*, and returns false otherwise. Similar operations (not shown in Figure 12.5(c)) are is_proper_subset_of(*S*), is_superset_of(*S*), and is_proper_superset_of(*S*). The **bag<*T*> type generator** allows duplicate elements in the collection and also inherits the Collection interface. It has three operations—create_union(b), create_intersection(b), and create_difference(b)—that all return a new object of type bag<*T*>.

A **list<*T*> type generator** inherits the Collection operations and can be used to create collections of objects of type T where the order of the elements is important. The value of each such object *O* is an *ordered list whose elements are of type T*. Hence, we can refer to the first, last, and *i*th element in the list. Also, when we add an element to the list, we must specify the position in the list where the element is inserted. Some of the list operations are shown in Figure 12.5(c). If *O* is an object of type list<*T*>, the operation *O*.insert_element_first(*E*) inserts the element *E* before the first element in the list *O*, so that *E* becomes the first element in the list. A similar operation (not shown) is *O*.insert_element_last(*E*). The operation *O*.insert_element_after(*E*, *I*) in Figure 12.5(c) inserts the element *E* after the *i*th element in the list *O* and will raise the exception InvalidIndex if no *i*th element exists in *O*. A similar operation (not shown) is *O*.insert_element_before(*E*, *I*). To remove elements from the list, the operations are *E* = *O*.remove_first_element(), *E* = *O*.remove_last_element(), and *E* = *O*.remove_element _at(*I*); these operations remove the indicated element from the list *and* return the element as the operation's result. Other operations retrieve an element without removing it from the list. These are *E* = *O*.retrieve_first_element(), *E* = *O*.retrieve _last_element(), and *E* = *O*.retrieve_element_at(*I*). Also, two operations to manipulate lists are defined. They are *P* = *O*.concat(*I*), which creates a new list *P* that is the concatenation of lists *O* and *I* (the elements in list *O* followed by those in list *I*), and *O*.append(*I*), which appends the elements of list *I* to the end of list *O* (without creating a new list object).

The **array<*T*> type generator** also inherits the Collection operations and is similar to list. Specific operations for an array object *O* are *O*.replace_element_at(*I*, *E*), which replaces the array element at position *I* with element *E*; *E* = *O*.remove_element_at(*I*), which retrieves the *i*th element and replaces it with a NULL value; and *E* = *O*.retrieve_element_at(*I*), which simply retrieves the *i*th element of the array. Any of these operations can raise the exception InvalidIndex if *I* is greater than the array's size. The operation *O*.resize(*N*) changes the number of array elements to *N*.

Figure 12.6
Inheritance hierarchy for the built-in interfaces of the object model.

The last type of collection objects are of type **dictionary**$<K,V>$. This allows the creation of a collection of association pairs $<K,V>$, where all K (key) values are unique. Making the key values unique allows for associative retrieval of a particular pair given its key value (similar to an index). If O is a collection object of type dictionary$<K,V>$, then O.bind(K,V) binds value V to the key K as an association $<K,V>$ in the collection, whereas O.unbind(K) removes the association with key K from O, and $V = O$.lookup(K) returns the value V associated with key K in O. The latter two operations can raise the exception KeyNotFound. Finally, O.contains_key(K) returns true if key K exists in O, and returns false otherwise.

Figure 12.6 is a diagram that illustrates the inheritance hierarchy of the built-in constructs of the object model. Operations are inherited from the supertype to the subtype. The collection interfaces described above are *not directly instantiable*; that is, one cannot directly create objects based on these interfaces. Rather, the interfaces can be used to generate user-defined collection types—of type set, bag, list, array, or dictionary—for a particular database application. If an attribute or class has a collection type, say a set, then it will inherit the operations of the set interface. For example, in a UNIVERSITY database application, the user can specify a type for set<STUDENT>, whose state would be sets of STUDENT objects. The programmer can then use the operations for set$<T>$ to manipulate an object of type set<STUDENT>. Creating application classes is typically done by utilizing the object definition language ODL (see Section 12.3.6).

It is important to note that all objects in a particular collection *must be of the same type.* Hence, although the keyword any appears in the specifications of collection interfaces in Figure 12.5(c), this does not mean that objects of any type can be intermixed within the same collection. Rather, it means that any type can be used when specifying the type of elements for a particular collection (including other collection types!).

12.3.4 Atomic (User-Defined) Objects

The previous section described the built-in collection types of the object model. Now we discuss how object types for *atomic objects* can be constructed. These are

specified using the keyword class in ODL. In the object model, any user-defined object that is not a collection object is called an **atomic object**.[27]

For example, in a UNIVERSITY database application, the user can specify an object type (class) for STUDENT objects. Most such objects will be **structured objects**; for example, a STUDENT object will have a complex structure, with many attributes, relationships, and operations, but it is still considered atomic because it is not a collection. Such a user-defined atomic object type is defined as a class by specifying its **properties** and **operations**. The properties define the state of the object and are further distinguished into **attributes** and **relationships**. In this subsection, we elaborate on the three types of components—attributes, relationships, and operations—that a user-defined object type for atomic (structured) objects can include. We illustrate our discussion with the two classes EMPLOYEE and DEPARTMENT shown in Figure 12.7.

An **attribute** is a property that describes some aspect of an object. Attributes have values (which are typically literals having a simple or complex structure) that are stored within the object. However, attribute values can also be Object_ids of other objects. Attribute values can even be specified via methods that are used to calculate the attribute value. In Figure 12.7[28] the attributes for EMPLOYEE are Name, Ssn, Birth_date, Sex, and Age, and those for DEPARTMENT are Dname, Dnumber, Mgr, Locations, and Projs. The Mgr and Projs attributes of DEPARTMENT have complex structure and are defined via **struct**, which corresponds to the *tuple constructor* of Section 12.1.3. Hence, the value of Mgr in each DEPARTMENT object will have two components: Manager, whose value is an Object_id that references the EMPLOYEE object that manages the DEPARTMENT, and Start_date, whose value is a date. The locations attribute of DEPARTMENT is defined via the set constructor, since each DEPARTMENT object can have a set of locations.

A **relationship** is a property that specifies that two objects in the database are related. In the object model of ODMG, only binary relationships (see Section 3.4) are explicitly represented, and each binary relationship is represented by a *pair of inverse references* specified via the keyword relationship. In Figure 12.7, one relationship exists that relates each EMPLOYEE to the DEPARTMENT in which he or she works—the Works_for relationship of EMPLOYEE. In the inverse direction, each DEPARTMENT is related to the set of EMPLOYEES that work in the DEPARTMENT—the Has_emps relationship of DEPARTMENT. The keyword **inverse** specifies that these two properties define a single conceptual relationship in inverse directions.[29]

By specifying inverses, the database system can maintain the referential integrity of the relationship automatically. That is, if the value of Works_for for a particular

---

[27]As mentioned earlier, this definition of *atomic object* in the ODMG object model is different from the definition of atom constructor given in Section 12.1.3, which is the definition used in much of the object-oriented database literature.

[28]We are using the Object Definition Language (ODL) notation in Figure 12.7, which will be discussed in more detail in Section 12.3.6.

[29]Section 7.4 discusses how a relationship can be represented by two attributes in inverse directions.

```
class EMPLOYEE
(    extent              ALL_EMPLOYEES
     key                 Ssn   )
{
     attribute           string              Name;
     attribute           string              Ssn;
     attribute           date Birth_date;
     attribute           enum Gender{M, F}   Sex;
     attribute           short               Age;
     relationship        DEPARTMENT          Works_for
                              inverse DEPARTMENT::Has_emps;
     void                reassign_emp(in string New_dname)
                              raises(dname_not_valid);
};
class DEPARTMENT
(    extent              ALL_DEPARTMENTS
     key                 Dname, Dnumber )
{
     attribute           string              Dname;
     attribute           short               Dnumber;
     attribute           struct Dept_mgr {EMPLOYEE Manager, date Start_date}
                              Mgr;
     attribute           set<string>         Locations;
     attribute           struct Projs {string Proj_name, time Weekly_hours)
                              Projs;
     relationship        set<EMPLOYEE>       Has_emps inverse EMPLOYEE::Works_for;
     void                add_emp(in string New_ename) raises(ename_not_valid);
     void                change_manager(in string New_mgr_name; in date
                              Start_date);
};
```

Figure 12.7

The attributes, relationships, and operations in a class definition.

EMPLOYEE *E* refers to DEPARTMENT *D*, then the value of Has_emps for DEPARTMENT *D* must include a reference to *E* in its set of EMPLOYEE references. If the database designer desires to have a relationship to be represented in *only one direction,* then it has to be modeled as an attribute (or operation). An example is the Manager component of the Mgr attribute in DEPARTMENT.

In addition to attributes and relationships, the designer can include **operations** in object type (class) specifications. Each object type can have a number of **operation signatures**, which specify the operation name, its argument types, and its returned value, if applicable. Operation names are unique within each object type, but they can be overloaded by having the same operation name appear in distinct object types. The operation signature can also specify the names of **exceptions** that can occur during operation execution. The implementation of the operation will include the code to raise these exceptions. In Figure 12.7 the EMPLOYEE class

has one operation: reassign_emp, and the DEPARTMENT class has two operations: add_emp and change_manager.

12.3.5 Extents, Keys, and Factory Objects

In the ODMG object model, the database designer can declare an *extent* (using the keyword **extent**) for any object type that is defined via a **class** declaration. The extent is given a name, and it will contain all persistent objects of that class. Hence, the extent behaves as a *set object* that holds all persistent objects of the class. In Figure 12.7 the EMPLOYEE and DEPARTMENT classes have extents called ALL_EMPLOYEES and ALL_DEPARTMENTS, respectively. This is similar to creating two objects—one of type set<EMPLOYEE> and the second of type set<DEPARTMENT>—and making them persistent by naming them ALL_EMPLOYEES and ALL_DEPARTMENTS. Extents are also used to automatically enforce the set/subset relationship between the extents of a supertype and its subtype. If two classes A and B have extents ALL_A and ALL_B, and class B is a subtype of class A (that is, class B extends class A), then the collection of objects in ALL_B must be a subset of those in ALL_A at any point. This constraint is automatically enforced by the database system.

A class with an extent can have one or more keys. A **key** consists of one or more properties (attributes or relationships) whose values are constrained to be unique for each object in the extent. For example, in Figure 12.7 the EMPLOYEE class has the Ssn attribute as key (each EMPLOYEE object in the extent must have a unique Ssn value), and the DEPARTMENT class has two distinct keys: Dname and Dnumber (each DEPARTMENT must have a unique Dname and a unique Dnumber). For a composite key[30] that is made of several properties, the properties that form the key are contained in parentheses. For example, if a class VEHICLE with an extent ALL_VEHICLES has a key made up of a combination of two attributes State and License_number, they would be placed in parentheses as (State, License_number) in the key declaration.

Next, we present the concept of **factory object**—an object that can be used to generate or create individual objects via its operations. Some of the interfaces of factory objects that are part of the ODMG object model are shown in Figure 12.8. The interface ObjectFactory has a single operation, new(), which returns a new object with an Object_id. By inheriting this interface, users can create their own factory interfaces for each user-defined (atomic) object type, and the programmer can implement the operation *new* differently for each type of object. Figure 12.8 also shows a DateFactory interface, which has additional operations for creating a new calendar_date and for creating an object whose value is the current_date, among other operations (not shown in Figure 12.8). As we can see, a factory object basically provides the **constructor operations** for new objects.

Finally, we discuss the concept of a **database**. Because an ODB system can create many different databases, each with its own schema, the ODMG object model has

---

[30]A composite key is called a *compound key* in the ODMG report.

```
interface ObjectFactory {
    Object        new();
};
interface SetFactory : ObjectFactory {
    Set           new_of_size(in long size);
};
interface ListFactory : ObjectFactory {
    List          new_of_size(in long size);
};
interface ArrayFactory : ObjectFactory {
    Array         new_of_size(in long size);
};
interface DictionaryFactory : ObjectFactory {
    Dictionary    new_of_size(in long size);
};
interface DateFactory : ObjectFactory {
    exception     InvalidDate{};
    …
    Date          calendar_date(    in unsigned short year,
                                    in unsigned short month,
                                    in unsigned short day )
                  raises(InvalidDate);
    …
    Date          current();
};
interface DatabaseFactory {
    Database      new();
};
interface Database {
    …
    void          open(in string database_name)
                        raises(DatabaseNotFound, DatabaseOpen);
    void          close() raises(DatabaseClosed, …);
    void          bind(in Object an_object, in string name)
                        raises(DatabaseClosed, ObjectNameNotUnique, …);
    Object        unbind(in string name)
                        raises(DatabaseClosed, ObjectNameNotFound, …);
    Object        lookup(in string object_name)
                        raises(DatabaseClosed, ObjectNameNotFound, …);
    … };
```

Figure 12.8
Interfaces to illustrate factory objects and database objects.

interfaces for DatabaseFactory and Database objects, as shown in Figure 12.8. Each database has its own *database name,* and the **bind** operation can be used to assign individual unique names to persistent objects in a particular database. The **lookup** operation returns an object from the database that has the specified persistent object_name, and the **unbind** operation removes the name of a persistent named object from the database.

12.3.6 The Object Definition Language ODL

After our overview of the ODMG object model in the previous section, we now show how these concepts can be utilized to create an object database schema using the object definition language ODL.[31]

The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular programming language. Its main use is to create object specifications—that is, classes and interfaces. Hence, ODL is not a programming language. A user can specify a database schema in ODL independently of any programming language, and then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, Smalltalk, and Java. We will give an overview of the C++ binding in Section 12.6.

Figure 12.9(b) shows a possible object schema for part of the UNIVERSITY database, which was presented in Chapter 4. We will describe the concepts of ODL using this example, and the one in Figure 12.11. The graphical notation for Figure 12.9(b) is shown in Figure 12.9(a) and can be considered as a variation of EER diagrams (see Chapter 4) with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships.

Figure 12.10 shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes. We will discuss these options further in Section 12.4.

Figure 12.10 shows the straightforward way of mapping part of the UNIVERSITY database from Chapter 4. Entity types are mapped into ODL classes, and inheritance is done using **extends**. However, there is no direct way to map categories (union types) or to do multiple inheritance. In Figure 12.10 the classes PERSON, FACULTY, STUDENT, and GRAD_STUDENT have the extents PERSONS, FACULTY, STUDENTS, and GRAD_STUDENTS, respectively. Both FACULTY and STUDENT **extends** PERSON and GRAD_STUDENT **extends** STUDENT. Hence, the collection of STUDENTS (and the collection of FACULTY) will be constrained to be a subset of the collection of PERSONs at any time. Similarly, the collection of

---

[31]The ODL syntax and data types are meant to be compatible with the Interface Definition language (IDL) of CORBA (Common Object Request Broker Architecture), with extensions for relationships and other database concepts.

(a)

Interface   Person-IF

Class   STUDENT

Relationships   1:1   1:N   M:N

Inheritance   Interface(is-a) inheritance using ":"   Class inheritance using **extends**

Figure 12.9
An example of a database schema.
(a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).

(b)

GRAD_STUDENTs will be a subset of STUDENTs. At the same time, individual STUDENT and FACULTY objects will inherit the properties (attributes and relationships) and operations of PERSON, and individual GRAD_STUDENT objects will inherit those of STUDENT.

The classes DEPARTMENT, COURSE, SECTION, and CURR_SECTION in Figure 12.10 are straightforward mappings of the corresponding entity types in Figure 12.9(b).

Figure 12.10
Possible ODL schema for the UNIVERSITY database in Figure 12.8(b).

```
class PERSON
(    extent          PERSONS
     key             Ssn )
{    attribute       struct Pname {   string    Fname,
                                      string    Mname,
                                      string    Lname }       Name;
     attribute       string                                   Ssn;
     attribute       date                                     Birth_date;
     attribute       enum Gender{M, F}                        Sex;
     attribute       struct Address {   short    No,
                                        string   Street,
                                        short    Apt_no,
                                        string   City,
                                        string   State,
                                        short    Zip }         Address;
     short           Age();   };
class FACULTY extends PERSON
(    extent          FACULTY )
{    attribute       string             Rank;
     attribute       float              Salary;
     attribute       string             Office;
     attribute       string             Phone;
     relationship    DEPARTMENT    Works_in inverse DEPARTMENT::Has_faculty;
     relationship    set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
     relationship    set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
     void            give_raise(in float raise);
     void            promote(in string new rank); };
class GRADE
(    extent GRADES )
{
     attribute       enum GradeValues{A,B,C,D,F,I, P} Grade;
     relationship    SECTION Section inverse SECTION::Students;
     relationship STUDENT Student inverse STUDENT::Completed_sections; };
class STUDENT extends PERSON
(    extent          STUDENTS )
{    attribute       string             Class;
     attribute       Department         Minors_in;
     relationship    Department Majors_in inverse DEPARTMENT::Has_majors;
     relationship    set<GRADE> Completed_sections inverse GRADE::Student;
     relationship    set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
     void            change_major(in string dname) raises(dname_not_valid);
     float           gpa();
     void            register(in short secno) raises(section_not_valid);
     void            assign_grade(in short secno; IN GradeValue grade)
                           raises(section_not_valid,grade_not_valid); };
```

Figure 12.10 (continued)
Possible ODL schema for the UNIVERSITY database in Figure 12.8(b).

```
class DEGREE
{    attribute      string            College;
     attribute      string            Degree;
     attribute      string            Year; };
class GRAD_STUDENT extends STUDENT
(    extent         GRAD_STUDENTS )
{    attribute      set<Degree>       Degrees;
     relationship   Faculty advisor inverse FACULTY::Advises;
     relationship   set<FACULTY>   Committee inverse FACULTY::On_committee_of;
     void           assign_advisor(in string Lname; in string Fname)
                          raises(faculty_not_valid);
     void           assign_committee_member(in string Lname; in string Fname)
                          raises(faculty_not_valid); };
class DEPARTMENT
(    extent         DEPARTMENTS
     key            Dname )
{    attribute      string            Dname;
     attribute      string            Dphone;
     attribute      string            Doffice;
     attribute      string            College;
     attribute      FACULTY           Chair;
     relationship   set<FACULTY> Has_faculty inverse FACULTY::Works_in;
     relationship   set<STUDENT> Has_majors inverse STUDENT::Majors_in;
     relationship   set<COURSE> Offers inverse COURSE::Offered_by; };
class COURSE
(    extent         COURSES
     key            Cno )
{    attribute      string            Cname;
     attribute      string            Cno;
     attribute      string            Description;
     relationship   set<SECTION> Has_sections inverse SECTION::Of_course;
     relationship   <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };
class SECTION
(    extent         SECTIONS )
{    attribute      short             Sec_no;
     attribute      string            Year;
     attribute      enum Quarter{Fall, Winter, Spring, Summer}
                          Qtr;
     relationship   set<Grade> Students inverse Grade::Section;
     relationship   COURSE Of_course inverse COURSE::Has_sections; };
class CURR_SECTION extends SECTION
(    extent         CURRENT_SECTIONS )
{    relationship   set<STUDENT> Registered_students
                          inverse STUDENT::Registered_in
     void           register_student(in string Ssn)
                          raises(student_not_valid, section_full); };
```

(a)

(b) interface GeometryObject
   {   attribute   enum   Shape{RECTANGLE, TRIANGLE, CIRCLE, … }
                                                     Shape;
        attribute   struct   Point {short x, short y} Reference_point;
        float   perimeter();
        float   area();
        void   translate(in short x_translation; in short y_translation);
        void   rotate(in float angle_of_rotation); };
   class RECTANGLE : GeometryObject
   (extent   RECTANGLES)
   {   attribute   struct   Point {short x, short y} Reference_point;
        attribute   short   Length;
        attribute   short   Height;
        attribute   float   Orientation_angle; };
   class TRIANGLE : GeometryObject
   (extent   TRIANGLES)
   {   attribute   struct   Point {short x, short y} Reference_point;
        attribute   short   Side_1;
        attribute   short   Side_2;
        attribute   float   Side1_side2_angle;
        attribute   float   Side1_orientation_angle; };
   class CIRCLE : GeometryObject
   (extent   CIRCLES)
   {   attribute   struct   Point {short x, short y} Reference_point;
        attribute   short   Radius; };
   …

Figure 12.11
An illustration of
interface inheritance
via ":". (a) Graphical
schema representation,
(b) Corresponding
interface and class
definitions in ODL.

However, the class GRADE requires some explanation. The GRADE class corresponds to the M:N relationship between STUDENT and SECTION in Figure 12.9(b). The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute Grade.[32]

Hence, the M:N relationship is mapped to the class GRADE, and a pair of 1:N relationships, one between STUDENT and GRADE and the other between SECTION and

---

[32]We will discuss alternative mappings for attributes of relationships in Section 12.4.

GRADE.[33] These relationships are represented by the following relationship properties: Completed_sections of STUDENT; Section and Student of GRADE; and Students of SECTION (see Figure 12.10). Finally, the class DEGREE is used to represent the composite, multivalued attribute degrees of GRAD_STUDENT (see Figure 8.10).

Because the previous example does not include any interfaces, only classes, we now utilize a different example to illustrate interfaces and interface (behavior) inheritance. Figure 12.11(a) is part of a database schema for storing geometric objects. An interface GeometryObject is specified, with operations to calculate the perimeter and area of a geometric object, plus operations to translate (move) and rotate an object. Several classes (RECTANGLE, TRIANGLE, CIRCLE, …) inherit the GeometryObject interface. Since GeometryObject is an interface, it is *noninstantiable*—that is, no objects can be created based on this interface directly. However, objects of type RECTANGLE, TRIANGLE, CIRCLE, … can be created, and these objects inherit all the operations of the GeometryObject interface. Note that with interface inheritance, only operations are inherited, not properties (attributes, relationships). Hence, if a property is needed in the inheriting class, it must be repeated in the class definition, as with the Reference_point attribute in Figure 12.11(b). Notice that the inherited operations can have different implementations in each class. For example, the implementations of the area and perimeter operations may be different for RECTANGLE, TRIANGLE, and CIRCLE.

Multiple inheritance of interfaces by a class is allowed, as is multiple inheritance of interfaces by another interface. However, with **extends** (class) inheritance, multiple inheritance is *not permitted.* Hence, a class can inherit via **extends** from at most one class (in addition to inheriting from zero or more interfaces).

12.4 Object Database Conceptual Design

Section 12.4.1 discusses how object database (ODB) design differs from relational database (RDB) design. Section 12.4.2 outlines a mapping algorithm that can be used to create an ODB schema, made of ODMG ODL class definitions, from a conceptual EER schema.

12.4.1 Differences between Conceptual Design of ODB and RDB

One of the main differences between ODB and RDB design is how relationships are handled. In ODB, relationships are typically handled by having relationship properties or reference attributes that include OID(s) of the related objects. These can be considered as *OID references* to the related objects. Both single references and collections of references are allowed. References for a binary relationship can be

---

[33]This is similar to how an M:N relationship is mapped in the relational model (see Section 9.1) and in the legacy network model (see Appendix E).

declared in a single direction, or in both directions, depending on the types of access expected. If declared in both directions, they may be specified as inverses of one another, thus enforcing the ODB equivalent of the relational referential integrity constraint.

In RDB, relationships among tuples (records) are specified by attributes with matching values. These can be considered as *value references* and are specified via *foreign keys,* which are values of primary key attributes repeated in tuples of the referencing relation. These are limited to being single-valued in each record because multivalued attributes are not permitted in the basic relational model. Thus, M:N relationships must be represented not directly, but as a separate relation (table), as discussed in Section 9.1.

Mapping binary relationships that contain attributes is not straightforward in ODBs, since the designer must choose in which direction the attributes should be included. If the attributes are included in both directions, then redundancy in storage will exist and may lead to inconsistent data. Hence, it is sometimes preferable to use the relational approach of creating a separate table by creating a separate class to represent the relationship. This approach can also be used for *n*-ary relationships, with degree *n* > 2.

Another major area of difference between ODB and RDB design is how inheritance is handled. In ODB, these structures are built into the model, so the mapping is achieved by using the inheritance constructs, such as *derived* (:) and **extends**. In relational design, as we discussed in Section 9.2, there are several options to choose from since no built-in construct exists for inheritance in the basic relational model. It is important to note, though, that object-relational and extended-relational systems are adding features to model these constructs directly as well as to include operation specifications in abstract data types (see Section 12.2).

The third major difference is that in ODB design, it is necessary to specify the operations early on in the design since they are part of the class specifications. Although it is important to specify operations during the design phase for all types of databases, the design of operations may be delayed in RDB design as it is not strictly required until the implementation phase.

There is a philosophical difference between the relational model and the object model of data in terms of behavioral specification. The relational model does *not* mandate the database designers to predefine a set of valid behaviors or operations, whereas this is a tacit requirement in the object model. One of the claimed advantages of the relational model is the support of ad hoc queries and transactions, whereas these are against the principle of encapsulation.

In practice, it is becoming commonplace to have database design teams apply object-based methodologies at early stages of conceptual design so that both the structure and the use or operations of the data are considered, and a complete specification is developed during conceptual design. These specifications are then mapped into relational schemas, constraints, and behavioral artifacts such as triggers or stored procedures (see Sections 5.2 and 13.4).

12.4.2 Mapping an EER Schema to an ODB Schema

It is relatively straightforward to design the type declarations of object classes for an ODBMS from an EER schema that contains *neither* categories *nor n*-ary relationships with $n > 2$. However, the operations of classes are not specified in the EER diagram and must be added to the class declarations after the structural mapping is completed. The outline of the mapping from EER to ODL is as follows:

Step 1. Create an ODL *class* for each EER entity type or subclass. The type of the ODL class should include all the attributes of the EER class.[34] *Multivalued attributes* are typically declared by using the set, bag, or list constructors.[35] If the values of the multivalued attribute for an object should be ordered, the list constructor is chosen; if duplicates are allowed, the bag constructor should be chosen; otherwise, the set constructor is chosen. *Composite attributes* are mapped into a tuple constructor (by using a struct declaration in ODL).

Declare an extent for each class, and specify any key attributes as keys of the extent.

Step 2. Add relationship properties or reference attributes for each *binary relationship* into the ODL classes that participate in the relationship. These may be created in one or both directions. If a binary relationship is represented by references in *both* directions, declare the references to be relationship properties that are inverses of one another, if such a facility exists.[36] If a binary relationship is represented by a reference in only *one* direction, declare the reference to be an attribute in the referencing class whose type is the referenced class name.

Depending on the cardinality ratio of the binary relationship, the relationship properties or reference attributes may be single-valued or collection types. They will be **single-valued** for binary relationships in the 1:1 or N:1 directions; they will be **collection types** (set-valued or list-valued[37]) for relationships in the 1:N or M:N direction. An alternative way to map binary M:N relationships is discussed in step 7.

If relationship attributes exist, a tuple constructor (struct) can be used to create a structure of the form <reference, relationship attributes>, which may be included instead of the reference attribute. However, this *does not allow the use of the inverse constraint*. Additionally, if this choice is represented in *both directions,* the attribute values will be represented twice, creating redundancy.

---

[34]This implicitly uses a tuple constructor at the top level of the type declaration, but in general, the tuple constructor is not explicitly shown in the ODL class declarations.

[35]Further analysis of the application domain is needed to decide which constructor to use because this information is not available from the EER schema.

[36]The ODL standard provides for the explicit definition of inverse relationships. Some ODBMS products may not provide this support; in such cases, programmers must maintain every relationship explicitly by coding the methods that update the objects appropriately.

[37]The decision whether to use set or list is not available from the EER schema and must be determined from the requirements.

Step 3. Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the original requirements. A constructor method should include program code that checks any constraints that must hold when a new object is created. A destructor method should check any constraints that may be violated when an object is deleted. Other methods should include any further constraint checks that are relevant.

Step 4. An ODL class that corresponds to a subclass in the EER schema inherits (via **extends**) the attributes, relationships, and methods of its superclass in the ODL schema. Its *specific* (local) attributes, relationship references, and operations are specified, as discussed in steps 1, 2, and 3.

Step 5. Weak entity types can be mapped in the same way as regular entity types. An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship; these can be mapped as though they were *composite multivalued attributes* of the owner entity type, by using the set<struct<…>> or list<struct<…>> constructors. The attributes of the weak entity are included in the struct<…> construct, which corresponds to a tuple constructor. Attributes are mapped as discussed in steps 1 and 2.

Step 6. Categories (union types) in an EER schema are difficult to map to ODL. It is possible to create a mapping similar to the EER-to-relational mapping (see Section 9.2) by declaring a class to represent the category and defining 1:1 relationships between the category and each of its superclasses.

Step 7. An *n*-ary relationship with degree *n* > 2 can be mapped into a separate class, with appropriate references to each participating class. These references are based on mapping a 1:N relationship from each class that represents a participating entity type to the class that represents the *n*-ary relationship. An M:N binary relationship, especially if it contains relationship attributes, may also use this mapping option, if desired.

The mapping has been applied to a subset of the UNIVERSITY database schema in Figure 4.10 in the context of the ODMG object database standard. The mapped object schema using the ODL notation is shown in Figure 12.10.

12.5 The Object Query Language OQL

The object query language OQL is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, Smalltalk, and Java. Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language. Additionally, the implementations of class operations in an ODMG schema can have their code written in these programming languages. The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

In Section 12.5.1 we will discuss the syntax of simple OQL queries and the concept of using named objects or extents as database entry points. Then, in Section 12.5.2 we will discuss the structure of query results and the use of path expressions to traverse relationships among objects. Other OQL features for handling object identity, inheritance, polymorphism, and other object-oriented concepts are discussed in Section 12.5.3. The examples to illustrate OQL queries are based on the UNIVERSITY database schema given in Figure 12.10.

12.5.1 Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a **select … from … where …** structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of 'Engineering' can be written as follows:

> **Q0: select** D.Dname
> **from** D **in** DEPARTMENTS
> **where** D.College = 'Engineering';

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object.* For many queries, the entry point is the name of the extent of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure 12.10, the named object DEPARTMENTS is of type set<DEPARTMENT>; PERSONS is of type set<PERSON>; FACULTY is of type set<FACULTY>; and so on.

The use of an extent name—DEPARTMENTS in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**[38]—D in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where clause. In Q0, only persistent objects D in the collection of DEPARTMENTS that satisfy the condition D.College = 'Engineering' are selected for the query result. For each selected object D, the value of D.Dname is retrieved in the query result. Hence, the *type of the result* for Q0 is bag<string> because the type of each Dname value is string (even though the actual result is a set because Dname is a key attribute). In general, the result of a query would be of type bag for select … from … and of type set for select distinct … from … , as in SQL (adding the keyword distinct eliminates duplicates).

Using the example in Q0, there are three syntactic options for specifying iterator variables:

> D **in** DEPARTMENTS
> DEPARTMENTS D
> DEPARTMENTS **AS** D

---

[38]This is similar to the tuple variables that range over tuples in SQL queries.

We will use the first construct in our examples.[39]

The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object, can be used as a database entry point.

12.5.2 Query Results and Path Expressions

In general, the result of a query can be of any type that can be expressed in the ODMG object model. A query does not have to follow the select … from … where … structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object. For example, the query

> **Q1:**   DEPARTMENTS;

returns a reference to the collection of all persistent DEPARTMENT objects, whose type is set<DEPARTMENT>. Similarly, suppose we had given (via the database bind operation, see Figure 12.8) a persistent name CS_DEPARTMENT to a single DEPARTMENT object (the Computer Science department); then, the query

> **Q1A:**   CS_DEPARTMENT;

returns a reference to that individual object of type DEPARTMENT. Once an entry point is specified, the concept of a **path expression** can be used to specify a *path* to related attributes and objects. A path expression typically starts at a *persistent object name,* or at the iterator variable that ranges over individual objects in a collection. This name will be followed by zero or more relationship names or attribute names connected using the *dot notation.* For example, referring to the UNIVERSITY database in Figure 12.10, the following are examples of path expressions, which are also valid queries in OQL:

> **Q2:**   CS_DEPARTMENT.Chair;
> **Q2A:**   CS_DEPARTMENT.Chair.Rank;
> **Q2B**:   CS_DEPARTMENT.Has_faculty;

The first expression Q2 returns an object of type FACULTY, because that is the type of the attribute Chair of the DEPARTMENT class. This will be a reference to the FACULTY object that is related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the attribute Chair; that is, a reference to the FACULTY object who is chairperson of the Computer Science department. The second expression Q2A is similar, except that it returns the Rank of this FACULTY object (the Computer Science chair) rather than the object reference; hence, the type returned by Q2A is string, which is the data type for the Rank attribute of the FACULTY class.

Path expressions Q2 and Q2A return single values, because the attributes Chair (of DEPARTMENT) and Rank (of FACULTY) are both single-valued and they are applied to a single object. The third expression, Q2B, is different; it returns an object of type set<FACULTY> even when applied to a single object, because that is the type of the relationship Has_faculty of the DEPARTMENT class. The collection returned will include

---

[39]Note that the latter two options are similar to the syntax for specifying tuple variables in SQL queries.

a set of references to all FACULTY objects that are related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the relationship Has_faculty; that is, a set of references to all FACULTY objects who are working in the Computer Science department. Now, to return the ranks of Computer Science faculty, we *cannot* write

Q3′:   CS_DEPARTMENT.Has_faculty.Rank;

because it is not clear whether the object returned would be of type set<string> or bag<string> (the latter being more likely, since multiple faculty may share the same rank). Because of this type of ambiguity problem, OQL does not allow expressions such as Q3′. Rather, one must use an iterator variable over any collections, as in Q3A or Q3B below:

Q3A: **select** F.Rank
             from F **in** CS_DEPARTMENT.Has_faculty;

Q3B: **select**   **distinct** F.Rank
             from F **in** CS_DEPARTMENT.Has_faculty;

Here, Q3A returns bag<string> (duplicate rank values appear in the result), whereas Q3B returns set<string> (duplicates are eliminated via the distinct keyword). Both Q3A and Q3B illustrate how an iterator variable can be defined in the from clause to range over a restricted collection specified in the query. The variable F in Q3A and Q3B ranges over the elements of the collection CS_DEPARTMENT.Has_faculty, which is of type set<FACULTY>, and includes only those faculty who are members of the Computer Science department.

In general, an OQL query can return a result with a complex structure specified in the query itself by utilizing the struct keyword. Consider the following examples:

Q4:   CS_DEPARTMENT.Chair.Advises;

Q4A: **select struct** (name: **struct** (last_name: S.name.Lname, first_name:
                                                    S.name.Fname),
                        degrees:(**select struct** (deg: D.Degree,
                                                    yr: D.Year,
                                                    college: D.College)
                                  from D **in** S.Degrees))
             from S **in** CS_DEPARTMENT.Chair.Advises;

Here, Q4 is straightforward, returning an object of type set<GRAD_STUDENT> as its result; this is the collection of graduate students who are advised by the chair of the Computer Science department. Now, suppose that a query is needed to retrieve the last and first names of these graduate students, plus the list of previous degrees of each. This can be written as in Q4A, where the variable S ranges over the collection of graduate students advised by the chairperson, and the variable D ranges over the degrees of each such student S. The type of the result of Q4A is a collection of (first-level) structs where each struct has two components: name and degrees.[40]

---

[40]As mentioned earlier, struct corresponds to the tuple constructor discussed in Section 12.1.3.

The name component is a further struct made up of last_name and first_name, each being a single string. The degrees component is defined by an embedded query and is itself a collection of further (second level) structs, each with three string components: deg, yr, and college.

Note that OQL is *orthogonal* with respect to specifying path expressions. That is, attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions, as long as the type system of OQL is not compromised. For example, one can write the following queries to retrieve the grade point average of all senior students majoring in Computer Science, with the result ordered by GPA, and within that by last and first name:

Q5A:	**select struct**	(last_name: S.name.Lname, first_name: S.name.Fname, gpa: S.gpa)
	from	S **in** CS_DEPARTMENT.Has_majors
	where	S.Class = 'senior'
	order by	gpa **desc**, last_name **asc**, first_name **asc**;

Q5B:	**select struct**	(last_name: S.name.Lname, first_name: S.name.Fname, gpa: S.gpa)
	from	S **in** STUDENTS
	where	S.Majors_in.Dname = 'Computer Science' **and** S.Class = 'senior'
	order by	gpa **desc**, last_name **asc**, first_name **asc**;

Q5A used the named entry point CS_DEPARTMENT to directly locate the reference to the Computer Science department and then locate the students via the relationship Has_majors, whereas Q5B searches the STUDENTS extent to locate all students majoring in that department. Notice how attribute names, relationship names, and operation (method) names are all used interchangeably (in an orthogonal manner) in the path expressions: gpa is an operation; Majors_in and Has_majors are relationships; and Class, Name, Dname, Lname, and Fname are attributes. The implementation of the gpa operation computes the grade point average and returns its value as a float type for each selected STUDENT.

The order by clause is similar to the corresponding SQL construct, and specifies in which order the query result is to be displayed. Hence, the collection returned by a query with an order by clause is of type *list*.

12.5.3 Other Features of OQL

Specifying Views as Named Queries. The view mechanism in OQL uses the concept of a **named query**. The **define** keyword is used to specify an identifier of the named query, which must be a unique name among all named objects, class names, method names, and function names in the schema. If the identifier has the same name as an existing named query, then the new definition replaces the previous definition. Once defined, a query definition is persistent until it is redefined or deleted. A view can also have parameters (arguments) in its definition.

For example, the following view V1 defines a named query Has_minors to retrieve the set of objects for students minoring in a given department:

V1:   **define**   Has_minors(Dept_name) **as**
      select   *S*
      from   *S* **in** STUDENTS
      where   *S*.Minors_in.Dname = Dept_name;

Because the ODL schema in Figure 12.10 only provided a unidirectional Minors_in attribute for a STUDENT, we can use the above view to represent its inverse without having to explicitly define a relationship. This type of view can be used to represent inverse relationships that are not expected to be used frequently. The user can now utilize the above view to write queries such as

Has_minors('Computer Science');

which would return a bag of students minoring in the Computer Science department. Note that in Figure 12.10, we defined Has_majors as an explicit relationship, presumably because it is expected to be used more often.

Extracting Single Elements from Singleton Collections. An OQL query will, in general, return a collection as its result, such as a bag, set (if distinct is specified), or list (if the order by clause is used). If the user requires that a query only return a single element, there is an **element** operator in OQL that is guaranteed to return a single element *E* from a singleton collection *C* that contains only one element. If *C* contains more than one element or if *C* is empty, then the element operator *raises an exception.* For example, Q6 returns the single object reference to the Computer Science department:

Q6:   **element** (**select** *D*
             from   *D* **in** DEPARTMENTS
             where *D*.Dname = 'Computer Science');

Since a department name is unique across all departments, the result should be one department. The type of the result is *D*:DEPARTMENT.

Collection Operators (Aggregate Functions, Quantifiers). Because many query expressions specify collections as their result, a number of operators have been defined that are applied to such collections. These include aggregate operators as well as membership and quantification (universal and existential) over a collection.

The aggregate operators (min, max, count, sum, avg) operate over a collection.[41] The operator count returns an integer type. The remaining aggregate operators (min, max, sum, avg) return the same type as the type of the operand collection. Two examples follow. The query Q7 returns the number of students minoring in Computer Science and Q8 returns the average GPA of all seniors majoring in Computer Science.

---

[41]These correspond to aggregate functions in SQL.

Q7: count (*S* **in** Has_minors('Computer Science'));

Q8: avg (**select**   *S*.Gpa
            from   *S* **in** STUDENTS
          where   *S*.Majors_in.Dname = 'Computer Science' **and**
                      S.Class = 'Senior');

Notice that aggregate operations can be applied to any collection of the appropriate type and can be used in any part of a query. For example, the query to retrieve all department names that have more than 100 majors can be written as in Q9:

Q9: select   *D*.Dname
       from   *D* **in** DEPARTMENTS
      where   **count** (*D*.Has_majors) > 100;

The *membership* and *quantification* expressions return a Boolean type—that is, true or false. Let *V* be a variable, *C* a collection expression, *B* an expression of type Boolean (that is, a Boolean condition), and *E* an element of the type of elements in collection *C*. Then:

(*E* **in** *C*) returns true if element *E* is a member of collection *C*.
(**for all** *V* **in** *C* : *B*) returns true if *all* the elements of collection *C* satisfy *B*.
(**exists** *V* **in** *C* : *B*) returns true if there is at least one element in *C* satisfying *B*.

To illustrate the membership condition, suppose we want to retrieve the names of all students who completed the course called 'Database Systems I'. This can be written as in Q10, where the nested query returns the collection of course names that each STUDENT *S* has completed, and the membership condition returns true if 'Database Systems I' is in the collection for a particular STUDENT *S*:

Q10: select   *S*.name.Lname, *S*.name.Fname
        from   *S* **in** STUDENTS
       where   'Database Systems I' **in**
               (**select**   *C*.Section.Of_course.Cname
                 from   *C* **in** *S*.Completed_sections);

Q10 also illustrates a simpler way to specify the select clause of queries that return a collection of structs; the type returned by Q10 is bag<struct(string, string)>.

One can also write queries that return true/false results. As an example, let us assume that there is a named object called JEREMY of type STUDENT. Then, query Q11 answers the following question: *Is Jeremy a Computer Science minor?* Similarly, Q12 answers the question *Are all Computer Science graduate students advised by Computer Science faculty?* Both Q11 and Q12 return true or false, which are interpreted as yes or no answers to the above questions:

Q11: JEREMY **in** Has_minors('Computer Science');

Q12: for all *G* **in**
          (**select** *S*
           from   *S* **in** GRAD_STUDENTS
           where *S*.Majors_in.Dname = 'Computer Science')
          : *G*.Advisor **in** CS_DEPARTMENT.Has_faculty;

Note that query Q12 also illustrates how attribute, relationship, and operation inheritance applies to queries. Although *S* is an iterator that ranges over the extent GRAD_STUDENTS, we can write *S*.Majors_in because the Majors_in relationship is inherited by GRAD_STUDENT from STUDENT via extends (see Figure 12.10). Finally, to illustrate the **exists** quantifier, query Q13 answers the following question: *Does any graduate Computer Science major have a 4.0 GPA?* Here, again, the operation gpa is inherited by GRAD_STUDENT from STUDENT via extends.

> Q13: **exists** *G* **in**
>   (**select** *S*
>   **from** *S* **in** GRAD_STUDENTS
>   **where** *S*.Majors_in.Dname = 'Computer Science')
>   : *G*.Gpa = 4;

Ordered (Indexed) Collection Expressions. As we discussed in Section 12.3.3, collections that are lists and arrays have additional operations, such as retrieving the *i*th, first, and last elements. Additionally, operations exist for extracting a sub-collection and concatenating two lists. Hence, query expressions that involve lists or arrays can invoke these operations. We will illustrate a few of these operations using sample queries. Q14 retrieves the last name of the faculty member who earns the highest salary:

> Q14: **first** (**select**   **struct**(facname: *F*.name.Lname, salary: *F*.Salary)
>   **from**   *F* **in** FACULTY
>   **order by**   salary **desc**);

Q14 illustrates the use of the **first** operator on a list collection that contains the salaries of faculty members sorted in descending order by salary. Thus, the first element in this sorted list contains the faculty member with the highest salary. This query assumes that only one faculty member earns the maximum salary. The next query, Q15, retrieves the top three Computer Science majors based on GPA.

> Q15: (**select**   **struct**(last_name: *S*.name.Lname, first_name: *S*.name.Fname,
>   gpa: *S*.Gpa)
>   **from**   *S* **in** CS_DEPARTMENT.Has_majors
>   **order by** gpa **desc**) **[0:2]**;

The select-from-order-by query returns a list of Computer Science students ordered by GPA in descending order. The first element of an ordered collection has an index position of 0, so the expression [0:2] returns a list containing the first, second, and third elements of the select … from … order by … result.

The Grouping Operator. The **group by** clause in OQL, although similar to the corresponding clause in SQL, provides explicit reference to the collection of objects within each *group* or *partition.* First we give an example, and then we describe the general form of these queries.

Q16 retrieves the number of majors in each department. In this query, the students are grouped into the same partition (group) if they have the same major; that is, the same value for *S*.Majors_in.Dname:

> **Q16:** (**select**   **struct**(dept_name, number_of_majors: **count** (**partition**))
>   **from** S **in** STUDENTS
>   **group by**   dept_name: S.Majors_in.Dname;

The result of the grouping specification is of type set<struct(dept_name: string, partition: bag<struct(S:STUDENT>)>), which contains a struct for each group (partition) that has two components: the grouping attribute value (dept_name) and the bag of the STUDENT objects in the group (partition). The select clause returns the grouping attribute (name of the department), and a count of the number of elements in each partition (that is, the number of students in each department), where **partition** is the keyword used to refer to each partition. The result type of the select clause is set<struct(dept_name: string, number_of_majors: integer)>. In general, the syntax for the group by clause is

> **group by** F_1: E_1, F_2: E_2, ... , F_k: E_k

where F_1: E_1, F_2: E_2, ... , F_k: E_k is a list of partitioning (grouping) attributes and each partitioning attribute specification F_i: E_i defines an attribute (field) name F_i and an expression E_i. The result of applying the grouping (specified in the group by clause) is a set of structures:

> set<struct(F_1: T_1, F_2: T_2, ... , F_k: T_k, partition: bag)>

where T_i is the type returned by the expression E_i, partition is a distinguished field name (a keyword), and B is a structure whose fields are the iterator variables (S in Q16) declared in the from clause having the appropriate type.

Just as in SQL, a **having** clause can be used to filter the partitioned sets (that is, select only some of the groups based on group conditions). In Q17, the previous query is modified to illustrate the having clause (and also shows the simplified syntax for the select clause). Q17 retrieves for each department having more than 100 majors, the average GPA of its majors. The having clause in Q17 selects only those partitions (groups) that have more than 100 elements (that is, departments with more than 100 students).

> **Q17:**   **select**   dept_name, avg_gpa: **avg** (**select** P.gpa **from** P **in partition**)
>   **from** S **in** STUDENTS
>   **group by**   dept_name: S.Majors_in.Dname
>   **having**   **count** (**partition**) > 100;

Note that the select clause of Q17 returns the average GPA of the students in the partition. The expression

> **select** P.Gpa **from** P **in partition**

returns a bag of student GPAs for that partition. The from clause declares an iterator variable P over the partition collection, which is of type bag<struct(S: STUDENT)>. Then the path expression P.gpa is used to access the GPA of each student in the partition.

12.6 Overview of the C++ Language Binding in the ODMG Standard

The C++ language binding specifies how ODL constructs are mapped to C++ constructs. This is done via a C++ class library that provides classes and operations that implement the ODL constructs. An object manipulation language (OML) is needed to specify how database objects are retrieved and manipulated within a C++ program, and this is based on the C++ programming language syntax and semantics. In addition to the ODL/OML bindings, a set of constructs called *physical pragmas* are defined to allow the programmer some control over physical storage issues, such as clustering of objects, utilizing indexes, and memory management.

The class library added to C++ for the ODMG standard uses the prefix d_ for class declarations that deal with database concepts.[42] The goal is that the programmer should think that only one language is being used, not two separate languages. For the programmer to refer to database objects in a program, a class D_Ref<*T*> is defined for each database class *T* in the schema. Hence, program variables of type D_Ref<*T*> can refer to both persistent and transient objects of class *T*.

In order to utilize the various built-in types in the ODMG object model such as collection types, various template classes are specified in the library. For example, an abstract class D_Object<*T*> specifies the operations to be inherited by all objects. Similarly, an abstract class D_Collection<*T*> specifies the operations of collections. These classes are not instantiable, but only specify the operations that can be inherited by all objects and by collection objects, respectively. A template class is specified for each type of collection; these include D_Set<*T*>, D_List<*T*>, D_Bag<*T*>, D_Varray<*T*>, and D_Dictionary<*T*>, and they correspond to the collection types in the object model (see Section 12.3.1). Hence, the programmer can create classes of types such as D_Set<D_Ref<STUDENT>> whose instances would be sets of references to STUDENT objects, or D_Set<string> whose instances would be sets of strings. Additionally, a class d_Iterator corresponds to the Iterator class of the object model.

The C++ ODL allows a user to specify the classes of a database schema using the constructs of C++ as well as the constructs provided by the object database library. For specifying the data types of attributes,[43] basic types such as d_Short (short integer), d_Ushort (unsigned short integer), d_Long (long integer), and d_Float (floating-point number) are provided. In addition to the basic data types, several structured literal types are provided to correspond to the structured literal types of the ODMG object model. These include d_String, d_Interval, d_Date, d_Time, and d_Timestamp (see Figure 12.5(b)).

---

[42]Presumably, d_ stands for *database* classes.

[43]That is, *member variables* in object-oriented programming terminology.

To specify relationships, the keyword rel_ is used within the prefix of type names; for example, by writing

d_Rel_Ref<DEPARTMENT, Has_majors> Majors_in;

in the STUDENT class, and

d_Rel_Set<STUDENT, Majors_in> Has_majors;

in the DEPARTMENT class, we are declaring that Majors_in and Has_majors are relationship properties that are inverses of one another and hence represent a 1:N binary relationship between DEPARTMENT and STUDENT.

For the OML, the binding overloads the operation *new* so that it can be used to create either persistent or transient objects. To create persistent objects, one must provide the database name and the persistent name of the object. For example, by writing

D_Ref<STUDENT> *S* = new(DB1, 'John_Smith') STUDENT;

the programmer creates a named persistent object of type STUDENT in database DB1 with persistent name John_Smith. Another operation, delete_object() can be used to delete objects. Object modification is done by the operations (methods) defined in each class by the programmer.

The C++ binding also allows the creation of extents by using the library class d_Extent. For example, by writing

D_Extent<PERSON> ALL_PERSONS(DB1);

the programmer would create a named collection object ALL_PERSONS—whose type would be D_Set<PERSON>—in the database DB1 that would hold persistent objects of type PERSON. However, key constraints are not supported in the C++ binding, and any key checks must be programmed in the class methods.[44] Also, the C++ binding does not support persistence via reachability; the object must be statically declared to be persistent at the time it is created.

12.7 Summary

In this chapter, we started in Section 12.1 with an overview of the concepts utilized in object databases, and we discussed how these concepts were derived from general object-oriented principles. The main concepts we discussed were: object identity and identifiers; encapsulation of operations; inheritance; complex structure of objects through nesting of type constructors; and how objects are made persistent.

---

[44]We have only provided a brief overview of the C++ binding. For full details, see Cattell et al. (2000), Chapter 5.

Then, in Section 12.2, we showed how many of these concepts were incorporated into the relational model and the SQL standard; we showed that this incorporation leads to expanded relational database functionality. These systems have been called object-relational databases.

We then discussed the ODMG 3.0 standard for object databases. We started by describing the various constructs of the object model in Sction 12.3. The various built-in types, such as Object, Collection, Iterator, set, list, and so on, were described by their interfaces, which specify the built-in operations of each type. These built-in types are the foundation upon which the object definition language (ODL) and object query language (OQL) are based. We also described the difference between objects, which have an ObjectId, and literals, which are values with no OID. Users can declare classes for their application that inherit operations from the appropriate built-in interfaces. Two types of properties can be specified in a user-defined class— attributes and relationships—in addition to the operations that can be applied to objects of the class. The ODL allows users to specify both interfaces and classes, and permits two different types of inheritance—interface inheritance via ":" and class inheritance via **extends**. A class can have an extent and keys. A description of ODL followed, and an example database schema for the UNIVERSITY database was used to illustrate the ODL constructs.

Following the description of the ODMG object model, we described a general technique for designing object database schemas in Section 12.4. We discussed how object databases differ from relational databases in three main areas: references to represent relationships, inclusion of operations, and inheritance. Finally, we showed how to map a conceptual database design in the EER model to the constructs of object databases.

In Section 12.5, we presented an overview of the object query language (OQL). The OQL follows the concept of orthogonality in constructing queries, meaning that an operation can be applied to the result of another operation as long as the type of the result is of the correct input type for the operation. The OQL syntax follows many of the constructs of SQL but includes additional concepts such as path expressions, inheritance, methods, relationships, and collections. Examples of how to use OQL over the UNIVERSITY database were given.

Next we gave an overview of the C++ language binding in Section 12.6, which extends C++ class declarations with the ODL type constructors but permits seamless integration of C++ with the ODBMS.

In 1997 Sun endorsed the ODMG API (Application Program Interface). O2 technologies was the first corporation to deliver an ODMG-compliant DBMS. Many ODBMS vendors, including Object Design (now eXcelon), Gemstone Systems, POET Software, and Versant Corporation[45], have endorsed the ODMG standard.

---

[45]The Versant Object Technology product now belongs to Actian Corporation.

Review Questions

12.1. What are the origins of the object-oriented approach?

12.2. What primary characteristics should an OID possess?

12.3. Discuss the various type constructors. How are they used to create complex object structures?

12.4. Discuss the concept of encapsulation, and tell how it is used to create abstract data types.

12.5. Explain what the following terms mean in object-oriented database terminology: *method, signature, message, collection, extent.*

12.6. What is the relationship between a type and its subtype in a type hierarchy? What is the constraint that is enforced on extents corresponding to types in the type hierarchy?

12.7. What is the difference between persistent and transient objects? How is persistence handled in typical OO database systems?

12.8. How do regular inheritance, multiple inheritance, and selective inheritance differ?

12.9. Discuss the concept of polymorphism/operator overloading.

12.10. Discuss how each of the following features is realized in SQL 2008: *object identifier, type inheritance, encapsulation of operations, and complex object structures.*

12.11. In the traditional relational model, creating a table defined both the table type (schema or attributes) and the table itself (extension or set of current tuples). How can these two concepts be separated in SQL 2008?

12.12. Describe the rules of inheritance in SQL 2008.

12.13. What are the differences and similarities between objects and literals in the ODMG object model?

12.14. List the basic operations of the following built-in interfaces of the ODMG object model: Object, Collection, Iterator, Set, List, Bag, Array, and Dictionary.

12.15. Describe the built-in structured literals of the ODMG object model and the operations of each.

12.16. What are the differences and similarities of attribute and relationship properties of a user-defined (atomic) class?

12.17. What are the differences and similarities of class inheritance via **extends** and interface inheritance via ":" in the ODMG object model?

12.18. Discuss how persistence is specified in the ODMG object model in the C++ binding.

12.19. Why are the concepts of extents and keys important in database applications?

12.20. Describe the following OQL concepts: *database entry points, path expressions, iterator variables, named queries (views), aggregate functions*, *grouping*, and *quantifiers.*

12.21. What is meant by the type orthogonality of OQL?

12.22. Discuss the general principles behind the C++ binding of the ODMG standard.

12.23. What are the main differences between designing a relational database and an object database?

12.24. Describe the steps of the algorithm for object database design by EER-to-OO mapping.

Exercises

12.25. Convert the example of GEOMETRY_OBJECTs given in Section 12.1.5 from the functional notation to the notation given in Figure 12.2 that distinguishes between attributes and operations. Use the keyword INHERIT to show that one class inherits from another class.

12.26. Compare inheritance in the EER model (see Chapter 4) to inheritance in the OO model described in Section 12.1.5.

12.27. Consider the UNIVERSITY EER schema in Figure 4.10. Think of what operations are needed for the entity types/classes in the schema. Do not consider constructor and destructor operations.

12.28. Consider the COMPANY ER schema in Figure 3.2. Think of what operations are needed for the entity types/classes in the schema. Do not consider constructor and destructor operations.

12.29. Design an OO schema for a database application that you are interested in. Construct an EER schema for the application, and then create the corresponding classes in ODL. Specify a number of methods for each class, and then specify queries in OQL for your database application.

12.30. Consider the AIRPORT database described in Exercise 4.21. Specify a number of operations/methods that you think should be applicable to that application. Specify the ODL classes and methods for the database.

12.31. Map the COMPANY ER schema in Figure 3.2 into ODL classes. Include appropriate methods for each class.

12.32. Specify in OQL the queries in the exercises of Chapters 6 and 7 that apply to the COMPANY database.

Selected Bibliography

Object-oriented database concepts are an amalgam of concepts from OO programming languages and from database systems and conceptual data models. A number of textbooks describe OO programming languages—for example, Stroustrup (1997) for C++, and Goldberg and Robson (1989) for Smalltalk. Books by Cattell (1994) and Lausen and Vossen (1997) describe OO database concepts. Other books on OO models include a detailed description of the experimental OODBMS developed at Microelectronic Computer Corporation called ORION and related OO topics by Kim and Lochovsky (1989). Bancilhon et al. (1992) describes the story of building the O2 OODBMS with a detailed discussion of design decisions and language implementation. Dogac et al. (1994) provides a thorough discussion on OO database topics by experts at a NATO workshop.

There is a vast bibliography on OO databases, so we can only provide a representative sample here. The October 1991 issue of *CACM* and the December 1990 issue of *ieee Computer* describe OO database concepts and systems. Dittrich (1986) and Zaniolo et al. (1986) survey the basic concepts of OO data models. An early paper on OO database system implementation is Baroody and DeWitt (1981). Su et al. (1988) presents an OO data model that was used in CAD/CAM applications. Gupta and Horowitz (1992) discusses OO applications to CAD, Network Management, and other areas. Mitschang (1989) extends the relational algebra to cover complex objects. Query languages and graphical user interfaces for OO are described in Gyssens et al. (1990), Kim (1989), Alashqur et al. (1989), Bertino et al. (1992), Agrawal et al. (1990), and Cruz (1992).

The Object-Oriented Manifesto by Atkinson et al. (1990) is an interesting article that reports on the position by a panel of experts regarding the mandatory and optional features of OO database management. Polymorphism in databases and OO programming languages is discussed in Osborn (1989), Atkinson and Buneman (1987), and Danforth and Tomlinson (1988). Object identity is discussed in Abiteboul and Kanellakis (1989). OO programming languages for databases are discussed in Kent (1991). Object constraints are discussed in Delcambre et al. (1991) and Elmasri, James, and Kouramajian (1993). Authorization and security in OO databases are examined in Rabitti et al. (1991) and Bertino (1992).

Cattell et al. (2000) describe the ODMG 3.0 standard, which is described in this chapter, and Cattell et al. (1993) and Cattell et al. (1997) describe the earlier versions of the standard. Bancilhon and Ferrari (1995) give a tutorial presentation of the important aspects of the ODMG standard. Several books describe the CORBA architecture—for example, Baker (1996).

The O2 system is described in Deux et al. (1991), and Bancilhon et al. (1992) includes a list of references to other publications describing various aspects of O2. The O2 model was formalized in Velez et al. (1989). The ObjectStore system

is described in Lamb et al. (1991). Fishman et al. (1987) and Wilkinson et al. (1990) discuss IRIS, an object-oriented DBMS developed at Hewlett-Packard Laboratories. Maier et al. (1986) and Butterworth et al. (1991) describe the design of GEMSTONE. The ODE system developed at AT&T Bell Labs is described in Agrawal and Gehani (1989). The ORION system developed at MCC is described in Kim et al. (1990). Morsi et al. (1992) describes an OO testbed.

Cattell (1991) surveys concepts from both relational and object databases and discusses several prototypes of object-based and extended relational database systems. Alagic (1997) points out discrepancies between the ODMG data model and its language bindings and proposes some solutions. Bertino and Guerrini (1998) propose an extension of the ODMG model for supporting composite objects. Alagic (1999) presents several data models belonging to the ODMG family.

This page intentionally left blank

XML: Extensible Markup Language

M any Internet applications provide Web interfaces to access information stored in one or more databases. These databases are often referred to as **data sources**. It is common to use the three-tier client/server architectures for Internet applications (see Section 2.5). Internet database applications are designed to interact with the user through Web interfaces that display Web pages on desktops, laptops, and mobile devices. The common method of specifying the contents and formatting of Web pages is through the use of **hypertext documents**. There are various languages for writing these documents, the most common being HTML (HyperText Markup Language). Although HTML is widely used for formatting and structuring Web *documents*, it is not suitable for specifying *structured data* that is extracted from databases. A new language—namely, XML (Extensible Markup Language)—has emerged as the standard for structuring and exchanging data over the Web in text files. Another language that can be used for the same purpose is JSON (JavaScript Object Notation; see Section 11.4). XML can be used to provide information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen. Both XML and JSON documents provide descriptive information, such as attribute names, as well as the values of these attributes, in a text file; hence, they are known as **self-describing documents**. The formatting aspects of Web pages are specified separately—for example, by using a formatting language such as XSL (Extensible Stylesheet Language) or a transformation language such as XSLT (Extensible Stylesheet Language for Transformations or simply XSL Transformations). Recently, XML has also been proposed as a possible model for data storage and retrieval, although only a few experimental database systems based on XML have been developed so far.

Basic HTML is useful for generating *static* Web pages with fixed text and other objects, but most e-commerce applications require Web pages that provide interactive features with the user and use the information provided by the user for selecting specific data from a database for display. Such Web pages are called *dynamic* Web pages, because the data extracted and displayed each time will be different depending on user input. For example, a banking app would get the user's account number, then extract the balance for that user's account from the database for display. We discussed how scripting languages, such as PHP, can be used to generate dynamic Web pages for applications such as those presented in Chapter 11. XML can be used to transfer information in self-describing textual files among various programs on different computers when needed by the applications.

In this chapter, we will focus on describing the XML data model and its associated languages, and how data extracted from relational databases can be formatted as XML documents to be exchanged over the Web. Section 13.1 discusses the difference among structured, semistructured, and unstructured data. Section 13.2 presents the XML data model, which is based on tree (hierarchical) structures as compared to the flat relational data model structures. In Section 13.3, we focus on the structure of XML documents and the languages for specifying the structure of these documents, such as DTD (Document Type Definition) and XML Schema. Section 13.4 shows the relationship between XML and relational databases. Section 13.5 describes some of the languages associated with XML, such as XPath and XQuery. Section 13.6 discusses how data extracted from relational databases can be formatted as XML documents. In Section 13.7, we discuss the new functions that have been incorporated into XML for the purpose of generating XML documents from relational databases. Finally, Section 13.8 is the chapter summary.

13.1 Structured, Semistructured, and Unstructured Data

The information stored in relational databases is known as **structured data** because it is represented in a strict format. For example, each record in a relational database table—such as each of the tables in the COMPANY database in Figure 5.6—follows the same format as the other records. For structured data, it is common to carefully design the database schema using techniques such as those described in Chapters 3 and 4 in order to define the database structure. The DBMS then checks to ensure that all data follows the structures and constraints specified in the schema.

However, not all data is collected and inserted into carefully designed structured databases. In some applications, data is collected in an ad hoc manner before it is known how it will be stored and managed. This data may have a certain structure, but not all the information collected will have the identical structure. Some attributes may be shared among the various entities, but other attributes may exist only in a few entities. Moreover, additional attributes can be introduced in some of the newer data items at any time, and there is no predefined schema. This type of data is known as **semistructured data**. A number of data models have been introduced

for representing semistructured data, often based on using tree or graph data structures rather than the flat relational model structures.

A key difference between structured and semistructured data concerns how the schema constructs (such as the names of attributes, relationships, and entity types) are handled. In semistructured data, the schema information is *mixed in* with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as **self-describing data**. Many of the newer NOSQL systems adopt self-describing storage schemes (see Chapter 24). Consider the following example. We want to collect a list of bibliographic references related to a certain research project. Some of these may be books or technical reports, others may be research articles in journals or conference proceedings, and still others may refer to complete journal issues or conference proceedings. Clearly, each of these may have different attributes and different types of information. Even for the same type of reference—say, conference articles—we may have different information. For example, one article citation may be complete, with full information about author names, title, proceedings, page numbers, and so on, whereas another citation may not have all the information available. New types of bibliographic sources may appear in the future—for instance, references to Web pages or to conference tutorials—and these may have new attributes that describe them.

One model for displaying semistructured data is a directed graph, as shown in Figure 13.1. The information shown in Figure 13.1 corresponds to some of the structured data shown in Figure 5.6. As we can see, this model somewhat resembles the object model (see Section 12.1.3) in its ability to represent complex objects and nested structures. In Figure 13.1, the **labels** or **tags** on the directed edges represent the schema names: the *names of attributes, object types* (or *entity types*

Figure 13.1
Representing semistructured data as a graph.

or *classes*), and *relationships*. The internal nodes represent individual objects or composite attributes. The leaf nodes represent actual data values of simple (atomic) attributes.

There are two main differences between the semistructured model and the object model that we discussed in Chapter 12:

1. The schema information—names of attributes, relationships, and classes (object types) in the semistructured model—is intermixed with the objects and their data values in the same data structure.

2. In the semistructured model, there is no requirement for a predefined schema to which the data objects must conform, although it is possible to define a schema if necessary. The object model of Chapter 12 requires a schema.

In addition to structured and semistructured data, a third category exists, known as **unstructured data** because there is very limited indication of the type of data. A typical example is a text document that contains information embedded within it. Web pages in HTML that contain some data are considered to be unstructured data. Consider part of an HTML file, shown in Figure 13.2. Text that appears between angled brackets, <…>, is an **HTML tag**. A tag with a slash, </…>, indicates an **end tag**, which represents the ending of the effect of a matching **start tag**. The tags **mark up** the document[1] in order to instruct an HTML processor how to display the text between a start tag and a matching end tag. Hence, the tags specify document formatting rather than the meaning of the various data elements in the document. HTML tags specify information, such as font size and style (boldface, italics, and so on), color, heading levels in documents, and so on. Some tags provide text structuring in documents, such as specifying a numbered or unnumbered list or a table. Even these structuring tags specify that the embedded textual data is to be displayed in a certain manner rather than indicating the type of data represented in the table.

HTML uses a large number of predefined tags, which are used to specify a variety of commands for formatting Web documents for display. The start and end tags specify the range of text to be formatted by each command. A few examples of the tags shown in Figure 13.2 follow:

- The <HTML> … </HTML> tags specify the boundaries of the document.
- The **document header** information—within the <HEAD> … </HEAD> tags—specifies various commands that will be used elsewhere in the document. For example, it may specify various **script functions** in a language such as JavaScript or PERL, or certain **formatting styles** (fonts, paragraph styles, header styles, and so on) that can be used in the document. It can also specify a title to indicate what the HTML file is for, and other similar information that will not be displayed as part of the document.

---

[1]That is why it is known as HyperText *Markup* Language.

```
<HTML>
    <HEAD>
    …
    </HEAD>
    <BODY>
        <H1>List of company projects and the employees in each project</H1>
        <H2>The ProductX project:</H2>
        <TABLE width="100%" border=0 cellpadding=0 cellspacing=0>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">John Smith:</FONT></TD>
                <TD>32.5 hours per week</TD>
            </TR>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">Joyce English:</FONT></TD>
                <TD>20.0 hours per week</TD>
            </TR>
        </TABLE>
        <H2>The ProductY project:</H2>
        <TABLE width="100%" border=0 cellpadding=0 cellspacing=0>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">John Smith:</FONT></TD>
                <TD>7.5 hours per week</TD>
            </TR>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">Joyce English:</FONT></TD>
                <TD>20.0 hours per week</TD>
            </TR>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">Franklin Wong:</FONT></TD>
                <TD>10.0 hours per week</TD>
            </TR>
        </TABLE>
    …
    </BODY>
</HTML>
```

Figure 13.2
Part of an HTML document
representing unstructured data.

- The **body** of the document—specified within the <BODY> … </BODY>
  tags—includes the document text and the markup tags that specify how the
  text is to be formatted and displayed. It can also include references to other
  objects, such as images, videos, voice messages, and other documents.
- The <H1> … </H1> tags specify that the text is to be displayed as a level 1
  heading. There are many heading levels (<H2>, <H3>, and so on), each
  displaying text in a less prominent heading format.
- The <TABLE> … </TABLE> tags specify that the following text is to be dis-
  played as a table. Each *table row* in the table is enclosed within <TR> … </TR>

tags, and the individual table data elements in a row are displayed within
<TD> … </TD> tags.[2]

- ■ Some tags may have **attributes**, which appear within the start tag and describe additional properties of the tag.[3]

In Figure 13.2, the <TABLE> start tag has four attributes describing various characteristics of the table. The following <TD> and <FONT> start tags have one and two attributes, respectively.

HTML has a very large number of predefined tags, and whole books are devoted to describing how to use these tags. If designed properly, HTML documents can be formatted so that humans are able to easily understand the document contents and are able to navigate through the resulting Web documents. However, the source HTML text documents are very difficult to interpret automatically by *computer programs* because they do not include schema information about the type of data in the documents. As e-commerce and other Internet applications become increasingly automated, it is becoming crucial to be able to exchange Web documents among various computer sites and to interpret their contents automatically. This need was one of the reasons that led to the development of XML. In addition, an extendible version of HTML called XHTML was developed that allows users to extend the tags of HTML for different applications and allows an XHTML file to be interpreted by standard XML processing programs. Our discussion will focus on XML only.

The example in Figure 13.2 illustrates a **static** HTML page, since all the information to be displayed is explicitly spelled out as fixed text in the HTML file. In many cases, some of the information to be displayed may be extracted from a database. For example, the project names and the employees working on each project may be extracted from the database in Figure 5.6 through the appropriate SQL query. We may want to use the same HTML formatting tags for displaying each project and the employees who work on it, but we may want to change the particular projects (and employees) being displayed. For example, we may want to see a Web page displaying the information for *ProjectX*, and then later a page displaying the information for *ProjectY*. Although both pages are displayed using the same HTML formatting tags, the actual data items displayed will be different. Such Web pages are called **dynamic**, since the data parts of the page may be different each time it is displayed, even though the display appearance is the same. We discussed in Chapter 11 how scripting languages, such as PHP, can be used to generate dynamic Web pages.

13.2 XML Hierarchical (Tree) Data Model

We now introduce the data model used in XML. The basic object in XML is the XML document. Two main structuring concepts are used to construct an XML document: **elements** and **attributes**. It is important to note that the term *attribute* in XML is *not*

---

[2]<TR> stands for table row and <TD> stands for table data.

[3]This is how the term *attribute* is used in document markup languages, which differs from how it is used in database models.

used in the same manner as is customary in database terminology, but rather as it is used in document description languages such as HTML and SGML.[4] Attributes in XML provide additional information that describes elements, as we will see. There are additional concepts in XML, such as entities, identifiers, and references, but first we concentrate on describing elements and attributes to show the essence of the XML model.

Figure 13.3 shows an example of an XML element called <Projects>. As in HTML, elements are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets < … >, and end tags are further identified by a slash, </ … >.[5]

Complex elements are constructed from other elements hierarchically, whereas **simple elements** contain data values. A major difference between XML and HTML is that XML tag names are defined to describe the meaning of the data elements in the document rather than to describe how the text is to be displayed. This makes it possible to process the data elements in the XML document automatically by computer programs. Also, the XML tag (element) names can be defined in another document, known as the *schema document*, to give a semantic meaning to the tag names that can be exchanged among multiple programs and users. In HTML, all tag names are predefined and fixed; that is why they are not extendible.

It is straightforward to see the correspondence between the XML textual representation shown in Figure 13.3 and the tree structure shown in Figure 13.1. In the tree representation, internal nodes represent complex elements, whereas leaf nodes represent simple elements. That is why the XML model is called a **tree model** or a **hierarchical model**. In Figure 13.3, the simple elements are the ones with the tag names <Name>, <Number>, <Location>, <Dept_no>, <Ssn>, <Last_name>, <First_name>, and <Hours>. The complex elements are the ones with the tag names <Projects>, <Project>, and <Worker>. In general, there is no limit on the levels of nesting of elements.

It is possible to characterize three main types of XML documents:

- **Data-centric XML documents.** These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over the Web. These usually follow a *predefined schema* that defines the tag names.
- **Document-centric XML documents.** These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.
- **Hybrid XML documents.** These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

---

[4]SGML (Standard Generalized Markup Language) is a more general language for describing documents and provides capabilities for specifying new tags. However, it is more complex than HTML and XML.

[5]The left and right angled bracket characters (< and >) are reserved characters, as are the ampersand (&), apostrophe ('), and single quotation mark ('). To include them within the text of a document, they must be encoded with escapes as <, >, &, ', and ", respectively.

```
<?xml version="1.0" standalone="yes"?>
    <Projects>
        <Project>
            <Name>ProductX</Name>
            <Number>1</Number>
            <Location>Bellaire</Location>
            <Dept_no>5</Dept_no>
            <Worker>
                <Ssn>123456789</Ssn>
                <Last_name>Smith</Last_name>
                <Hours>32.5</Hours>
            </Worker>
            <Worker>
                <Ssn>453453453</Ssn>
                <First_name>Joyce</First_name>
                <Hours>20.0</Hours>
            </Worker>
        </Project>
        <Project>
            <Name>ProductY</Name>
            <Number>2</Number>
            <Location>Sugarland</Location>
            <Dept_no>5</Dept_no>
            <Worker>
                <Ssn>123456789</Ssn>
                <Hours>7.5</Hours>
            </Worker>
            <Worker>
                <Ssn>453453453</Ssn>
                <Hours>20.0</Hours>
            </Worker>
            <Worker>
                <Ssn>333445555</Ssn>
                <Hours>10.0</Hours>
            </Worker>
        </Project>
        …
    </Projects>
```

Figure 13.3

A complex XML element called <Projects>.

XML documents that do not follow a predefined schema of element names and corresponding tree structure are known as **schemaless XML documents**. It is important to note that data-centric XML documents can be considered either as semistructured data or as structured data as defined in Section 13.1. If an XML document conforms to a predefined XML schema or DTD (see Section 13.3), then the document can be considered as *structured data*. On the other hand, XML allows

documents that do not conform to any schema; these would be considered as *semistructured data* and are *schemaless XML documents*. When the value of the standalone attribute in an XML document is yes, as in the first line in Figure 13.3, the document is standalone and schemaless.

XML attributes are generally used in a manner similar to how they are used in HTML (see Figure 13.2), namely, to describe properties and characteristics of the elements (tags) within which they appear. It is also possible to use XML attributes to hold the values of simple data elements; however, this is generally not recommended. An exception to this rule is in cases that need to **reference** another element in another part of the XML document. To do this, it is common to use attribute values in one element as the references. This resembles the concept of foreign keys in relational databases, and it is a way to get around the strict hierarchical model that the XML tree model implies. We discuss XML attributes further in Section 13.3 when we discuss XML schema and DTD.

13.3 XML Documents, DTD, and XML Schema

13.3.1 Well-Formed and Valid XML Documents and XML DTD

In Figure 13.3, we saw what a simple XML document may look like. An XML document is **well formed** if it follows a few conditions. In particular, it must start with an **XML declaration** to indicate the version of XML being used as well as any other relevant attributes, as shown in the first line in Figure 13.3. It must also follow the syntactic guidelines of the tree data model. This means that there should be a *single root element*, and every element must include a matching pair of start and end tags *within* the start and end tags *of the parent element*. This ensures that the nested elements specify a well-formed tree structure.

A well-formed XML document is syntactically correct. This allows it to be processed by generic processors that traverse the document and create an internal tree representation. A standard model with an associated set of API (application programming interface) functions called **DOM** (Document Object Model) allows programs to manipulate the resulting tree representation corresponding to a well-formed XML document. However, the whole document must be parsed beforehand when using DOM in order to convert the document to that standard DOM internal data structure representation. Another API called **SAX** (Simple API for XML) allows processing of XML documents on the fly by notifying the processing program through callbacks whenever a start or end tag is encountered. This makes it easier to process large documents and allows for processing of so-called **streaming XML documents**, where the processing program can process the tags as they are encountered. This is also known as **event-based processing**. There are also other specialized processors that work with various programming and scripting languages for parsing XML documents.

A well-formed XML document can be schemaless; that is, it can have any tag names for the elements within the document. In this case, there is no predefined

set of elements (tag names) that a program processing the document knows to expect. This gives the document creator the freedom to specify new elements but limits the possibilities for automatically interpreting the meaning or semantics of the elements within the document.

A stronger criterion is for an XML document to be **valid**. In this case, the document must be well formed, and it must follow a particular schema. That is, the element names used in the start and end tag pairs must follow the structure specified in a separate XML **DTD** (**Document Type Definition**) file or **XML schema file**. We first discuss XML DTD here, and then we give an overview of XML schema in Section 13.3.2. Figure 13.4 shows a simple XML DTD file, which specifies the elements (tag names) and their nested structures. Any valid documents conforming to this DTD should follow the specified structure. A special syntax exists for specifying DTD files, as illustrated in Figure 13.4(a). First, a name is given to the **root tag** of the document, which is called Projects in the first line in Figure 13.4. Then the elements and their nested structure are specified.

When specifying elements, the following notation is used:

- A * following the element name means that the element can be repeated zero or more times in the document. This kind of element is known as an *optional multivalued (repeating) element.*
- A + following the element name means that the element can be repeated one or more times in the document. This kind of element is a *required multivalued (repeating) element.*
- A ? following the element name means that the element can be repeated zero or one times. This kind is an *optional single-valued (nonrepeating) element.*
- An element appearing without any of the preceding three symbols must appear exactly once in the document. This kind is a *required single-valued (nonrepeating) element.*
- The **type** of the element is specified via parentheses following the element. If the parentheses include names of other elements, these latter elements are the *children* of the element in the tree structure. If the parentheses include the keyword #PCDATA or one of the other data types available in XML DTD, the element is a leaf node. PCDATA stands for *parsed character data*, which is roughly similar to a string data type.
- The list of attributes that can appear within an element can also be specified via the keyword !ATTLIST. In Figure 13.3, the Project element has an attribute ProjId. If the type of an attribute is ID, then it can be referenced from another attribute whose type is IDREF within another element. Notice that attributes can also be used to hold the values of simple data elements of type #PCDATA.
- Parentheses can be nested when specifying elements.
- A bar symbol ($e_1 \mid e_2$) specifies that either e_1 or e_2 can appear in the document.

We can see that the tree structure in Figure 13.1 and the XML document in Figure 13.3 conform to the XML DTD in Figure 13.4. To require that an XML document be checked for conformance to a DTD, we must specify this in the

(a) <!DOCTYPE Projects [
    <!ELEMENT Projects (Project+)>
    <!ELEMENT Project (Name, Number, Location, Dept_no?, Workers)>
        <!ATTLIST Project
            ProjId ID #REQUIRED>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Number (#PCDATA)
    <!ELEMENT Location (#PCDATA)>
    <!ELEMENT Dept_no (#PCDATA)>
    <!ELEMENT Workers (Worker*)>
    <!ELEMENT Worker (Ssn, Last_name?, First_name?, Hours)>
    <!ELEMENT Ssn (#PCDATA)>
    <!ELEMENT Last_name (#PCDATA)>
    <!ELEMENT First_name (#PCDATA)>
    <!ELEMENT Hours (#PCDATA)>
  ] >

(b) <!DOCTYPE Company [
    <!ELEMENT Company((Employee|Department|Project)*)>
    <!ELEMENT Department (DName, Location+)>
        <!ATTLIST Department
            DeptId ID #REQUIRED>

    <!ELEMENT Employee (EName, Job, Salary)>
        <!ATTLIST Project
            EmpId ID #REQUIRED
            DeptId IDREF #REQUIRED>
    <!ELEMENT Project (PName, Location)
        <!ATTLIST Project
            ProjId ID #REQUIRED
            Workers IDREFS #IMPLIED>
    <!ELEMENT DName (#PCDATA)>
    <!ELEMENT EName (#PCDATA)>
    <!ELEMENT PName (#PCDATA)>
    <!ELEMENT Job (#PCDATA)
    <!ELEMENT Location (#PCDATA)>
    <!ELEMENT Salary (#PCDATA)>
  ] >

Figure 13.4
(a) An XML DTD file called *Projects*. (b) An XML DTD file called *Company.*

declaration of the document. For example, we could change the first line in Figure 13.3 to the following:

    <?xml version = "1.0" standalone = "no"?>
    <!DOCTYPE Projects SYSTEM "proj.dtd">

When the value of the standalone attribute in an XML document is "no", the document needs to be checked against a separate DTD document or XML schema document (see Section 13.2.2). The DTD file shown in Figure 13.4 should be stored in

the same file system as the XML document and should be given the file name proj.dtd. Alternatively, we could include the DTD document text at the beginning of the XML document itself to allow the checking.

Figure 13.4(b) shows another DTD document called Company to illustrate the use of IDREF. A Company document can have any number of Department, Employee, and Project elements, with IDs DeptID, EmpId, and ProjID, respectively. The Employee element has an attribute DeptId of type IDREF, which is a reference to the Department element where the employee works; this is similar to a foreign key. The Project element has an attribute Workers of type IDREFS, which will hold a list of Employee EmpIDs that work on that project; this is similar to a collection or list of foreign keys. The #IMPLIED keyword means that this attribute is optional. It is also possible to provide a default value for any attribute.

Although XML DTD is adequate for specifying tree structures with required, optional, and repeating elements, and with various types of attributes, it has several limitations. First, the data types in DTD are not very general. Second, DTD has its own special syntax and thus requires specialized processors. It would be advantageous to specify XML schema documents using the syntax rules of XML itself so that the same processors used for XML documents could process XML schema descriptions. Third, all DTD elements are always forced to follow the specified ordering of the document, so unordered elements are not permitted. These drawbacks led to the development of XML schema, a more general but also more complex language for specifying the structure and elements of XML documents.

13.3.2 XML Schema

The **XML schema language** is a standard for specifying the structure of XML documents. It uses the same syntax rules as regular XML documents, so that the same processors can be used on both. To distinguish the two types of documents, we will use the term *XML instance document* or *XML document* for a regular XML document that contains both tag names and data values, and *XML schema document* for a document that specifies an XML schema. An XML schema document would contain only tag names, tree structure information, constraints, and other descriptions but no data values. Figure 13.5 shows an XML schema document corresponding to the COMPANY database shown in Figure 5.5. Although it is unlikely that we would want to display the whole database as a single document, there have been proposals to store data in *native XML* format as an alternative to storing the data in relational databases. The schema in Figure 13.5 would serve the purpose of specifying the structure of the COMPANY database if it were stored in a native XML system. We discuss this topic further in Section 13.4.

As with XML DTD, XML schema is based on the tree data model, with elements and attributes as the main structuring concepts. However, it borrows additional concepts from database and object models, such as keys, references, and identifiers. Here we describe the features of XML schema in a step-by-step manner, referring to the sample XML schema document in Figure 13.5 for illustration. We introduce and describe some of the schema concepts in the order in which they are used in Figure 13.5.

Figure 13.5
An XML schema file called *company*.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">Company Schema (Element Approach) - Prepared by Babak
            Hojabri</xsd:documentation>
    </xsd:annotation>
<xsd:element name="company">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="department" type="Department" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="employee" type="Employee" minOccurs="0" maxOccurs="unbounded">
                <xsd:unique name="dependentNameUnique">
                    <xsd:selector xpath="employeeDependent" />
                    <xsd:field xpath="dependentName" />
                </xsd:unique>
            </xsd:element>
            <xsd:element name="project" type="Project" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:unique name="departmentNameUnique">
        <xsd:selector xpath="department" />
        <xsd:field xpath="departmentName" />
    </xsd:unique>
    <xsd:unique name="projectNameUnique">
        <xsd:selector xpath="project" />
        <xsd:field xpath="projectName" />
    </xsd:unique>
    <xsd:key name="projectNumberKey">
        <xsd:selector xpath="project" />
        <xsd:field xpath="projectNumber" />
    </xsd:key>
    <xsd:key name="departmentNumberKey">
        <xsd:selector xpath="department" />
        <xsd:field xpath="departmentNumber" />
    </xsd:key>
    <xsd:key name="employeeSSNKey">
        <xsd:selector xpath="employee" />
        <xsd:field xpath="employeeSSN" />
    </xsd:key>
    <xsd:keyref name="departmentManagerSSNKeyRef" refer="employeeSSNKey">
        <xsd:selector xpath="department" />
        <xsd:field xpath="departmentManagerSSN" />
    </xsd:keyref>
```

(continues)

Figure 13.5 (continued)
An XML schema file called *company*.

```
    <xsd:keyref name="employeeDepartmentNumberKeyRef"
        refer="departmentNumberKey">
        <xsd:selector xpath="employee" />
        <xsd:field xpath="employeeDepartmentNumber" />
    </xsd:keyref>
    <xsd:keyref name="employeeSupervisorSSNKeyRef" refer="employeeSSNKey">
        <xsd:selector xpath="employee" />
        <xsd:field xpath="employeeSupervisorSSN" />
    </xsd:keyref>
    <xsd:keyref name="projectDepartmentNumberKeyRef" refer="departmentNumberKey">
        <xsd:selector xpath="project" />
        <xsd:field xpath="projectDepartmentNumber" />
    </xsd:keyref>
    <xsd:keyref name="projectWorkerSSNKeyRef" refer="employeeSSNKey">
        <xsd:selector xpath="project/projectWorker" />
        <xsd:field xpath="SSN" />
    </xsd:keyref>
    <xsd:keyref name="employeeWorksOnProjectNumberKeyRef"
        refer="projectNumberKey">
        <xsd:selector xpath="employee/employeeWorksOn" />
        <xsd:field xpath="projectNumber" />
    </xsd:keyref>
</xsd:element>
<xsd:complexType name="Department">
    <xsd:sequence>
        <xsd:element name="departmentName" type="xsd:string" />
        <xsd:element name="departmentNumber" type="xsd:string" />
        <xsd:element name="departmentManagerSSN" type="xsd:string" />
        <xsd:element name="departmentManagerStartDate" type="xsd:date" />
        <xsd:element name="departmentLocation" type="xsd:string" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Employee">
    <xsd:sequence>
        <xsd:element name="employeeName" type="Name" />
        <xsd:element name="employeeSSN" type="xsd:string" />
        <xsd:element name="employeeSex" type="xsd:string" />
        <xsd:element name="employeeSalary" type="xsd:unsignedInt" />
        <xsd:element name="employeeBirthDate" type="xsd:date" />
        <xsd:element name="employeeDepartmentNumber" type="xsd:string" />
        <xsd:element name="employeeSupervisorSSN" type="xsd:string" />
        <xsd:element name="employeeAddress" type="Address" />
        <xsd:element name="employeeWorksOn" type="WorksOn" minOccurs="1" maxOccurs="unbounded" />
        <xsd:element name="employeeDependent" type="Dependent" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
```

Figure 13.5 (continued)
An XML schema file called *company*.

```
<xsd:complexType name="Project">
    <xsd:sequence>
        <xsd:element name="projectName" type="xsd:string" />
        <xsd:element name="projectNumber" type="xsd:string" />
        <xsd:element name="projectLocation" type="xsd:string" />
        <xsd:element name="projectDepartmentNumber" type="xsd:string" />
        <xsd:element name="projectWorker" type="Worker" minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Dependent">
    <xsd:sequence>
        <xsd:element name="dependentName" type="xsd:string" />
        <xsd:element name="dependentSex" type="xsd:string" />
        <xsd:element name="dependentBirthDate" type="xsd:date" />
        <xsd:element name="dependentRelationship" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Address">
    <xsd:sequence>
        <xsd:element name="number" type="xsd:string" />
        <xsd:element name="street" type="xsd:string" />
        <xsd:element name="city" type="xsd:string" />
        <xsd:element name="state" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Name">
    <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string" />
        <xsd:element name="middleName" type="xsd:string" />
        <xsd:element name="lastName" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Worker">
    <xsd:sequence>
        <xsd:element name="SSN" type="xsd:string" />
        <xsd:element name="hours" type="xsd:float" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="WorksOn">
    <xsd:sequence>
        <xsd:element name="projectNumber" type="xsd:string" />
        <xsd:element name="hours" type="xsd:float" />
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

1. **Schema descriptions and XML namespaces.** It is necessary to identify the specific set of XML schema language elements (tags) being used by specifying a file stored at a Web site location. The second line in Figure 13.5 specifies the file used in this example, which is http://www.w3.org/2001/XMLSchema. This is a commonly used standard for XML schema commands. Each such definition is called an **XML namespace** because it defines the set of commands (names) that can be used. The file name is assigned to the variable xsd (XML schema description) using the attribute xmlns (XML namespace), and this variable is used as a prefix to all XML schema commands (tag names). For example, in Figure 13.5, when we write xsd:element or xsd:sequence, we are referring to the definitions of the element and sequence tags as defined in the file http://www.w3.org/2001/XMLSchema.

2. **Annotations, documentation, and language used.** The next couple of lines in Figure 13.5 illustrate the XML schema elements (tags) xsd:annotation and xsd:documentation, which are used for providing comments and other descriptions in the XML document. The attribute xml:lang of the xsd:documentation element specifies the language being used, where en stands for the English language.

3. **Elements and types.** Next, we specify the *root element* of our XML schema. In XML schema, the name attribute of the xsd:element tag specifies the element name, which is called company for the root element in our example (see Figure 13.5). The structure of the company root element can then be specified, which in our example is xsd:complexType. This is further specified to be a sequence of departments, employees, and projects using the xsd:sequence structure of XML schema. It is important to note here that this is not the only way to specify an XML schema for the COMPANY database. We will discuss other options in Section 13.6.

4. **First-level elements in the COMPANY database.** Next, we specify the three first-level elements under the company root element in Figure 13.5. These elements are named employee, department, and project, and each is specified in an xsd:element tag. Notice that if a tag has only attributes and no further subelements or data within it, it can be ended with the backslash symbol (/>) directly instead of having a separate matching end tag. These are called **empty elements**; examples are the xsd:element elements named department and project in Figure 13.5.

5. **Specifying element type and minimum and maximum occurrences.** In XML schema, the attributes type, minOccurs, and maxOccurs in the xsd:element tag specify the type and multiplicity of each element in any document that conforms to the schema specifications. If we specify a type attribute in an xsd:element, the structure of the element must be described separately, typically using the xsd:complexType element of XML schema. This is illustrated by the employee, department, and project elements in Figure 13.5. On the other hand, if no type attribute is specified, the element structure can be defined directly following the tag, as illustrated by the company root element in Figure 13.5. The minOccurs and maxOccurs tags are used for specifying lower

and upper bounds on the number of occurrences of an element in any XML document that conforms to the schema specifications. If they are not specified, the default is exactly one occurrence. These serve a similar role to the *, +, and ? symbols of XML DTD.

6. **Specifying keys.** In XML schema, it is possible to specify constraints that correspond to unique and primary key constraints in a relational database (see Section 5.2.2), as well as foreign keys (or referential integrity) constraints (see Section 5.2.4). The xsd:unique tag specifies elements that correspond to unique attributes in a relational database. We can give each such uniqueness constraint a name, and we must specify xsd:selector and xsd:field tags for it to identify the element type that contains the unique element and the element name within it that is unique via the xpath attribute. This is illustrated by the departmentNameUnique and projectNameUnique elements in Figure 13.5. For specifying **primary keys**, the tag xsd:key is used instead of xsd:unique, as illustrated by the projectNumberKey, department-NumberKey, and employeeSSNKey elements in Figure 13.5. For specifying **foreign keys**, the tag xsd:keyref is used, as illustrated by the six xsd:keyref elements in Figure 13.5. When specifying a foreign key, the attribute refer of the xsd:keyref tag specifies the referenced primary key, whereas the tags xsd:selector and xsd:field specify the referencing element type and foreign key (see Figure 13.5).

7. **Specifying the structures of complex elements via complex types.** The next part of our example specifies the structures of the complex elements Department, Employee, Project, and Dependent, using the tag xsd:complexType (see Figure 13.5). We specify each of these as a sequence of subelements corresponding to the database attributes of each entity type (see Figure 7.7)by using the xsd:sequence and xsd:element tags of XML schema. Each element is given a name and type via the attributes name and type of xsd:element. We can also specify minOccurs and maxOccurs attributes if we need to change the default of exactly one occurrence. For (optional) database attributes where null is allowed, we need to specify minOccurs = 0, whereas for multivalued database attributes we need to specify maxOccurs = "unbounded" on the corresponding element. Notice that if we were not going to specify any key constraints, we could have embedded the subelements within the parent element definitions directly without having to specify complex types. However, when unique, primary key and foreign key constraints need to be specified; we must define complex types to specify the element structures.

8. **Composite (compound) attributes.** Composite attributes from Figure 9.2 are also specified as complex types in Figure 13.7, as illustrated by the Address, Name, Worker, and WorksOn complex types. These could have been directly embedded within their parent elements.

This example illustrates some of the main features of XML schema. There are other features, but they are beyond the scope of our presentation. In the next section, we discuss the different approaches to creating XML documents from relational databases and storing XML documents.

13.4 Storing and Extracting XML Documents from Databases

Several approaches to organizing the contents of XML documents to facilitate their subsequent querying and retrieval have been proposed. The following are the most common approaches:

1. **Using a file system or a DBMS to store the documents as text.** An XML document can be stored as a text file within a traditional file system. Alternatively, a relational DBMS can be used to store whole XML documents as text fields within the DBMS recordss. This approach can be used if the DBMS has a special module for document processing, and it would work for storing schemaless and document-centric XML documents.

2. **Using a DBMS to store the document contents as data elements.** This approach would work for storing a collection of documents that follow a specific XML DTD or XML schema. Because all the documents have the same structure, one can design a relational database to store the leaf-level data elements within the XML documents. This approach would require mapping algorithms to design a database schema that is compatible with the XML document structure as specified in the XML schema or DTD and to re-create the XML documents from the stored data. These algorithms can be implemented either as an internal DBMS module or as separate middleware that is not part of the DBMS. If all elements in an XML document have IDs, a simple representation would be to have a table with attributes XDOC(CId, PId, Etag, Val) where CID and PId are the parent and child element IDs, Etag is the name of the element of the Cid, and Val is the value if it is a leaf node, assuming all values are the same type.

3. **Designing a specialized system for storing native XML data.** A new type of database system based on the hierarchical (tree) model could be designed and implemented. Such systems are referred to as **native XML DBMS**s. The system would include specialized indexing and querying techniques and would work for all types of XML documents. It could also include data compression techniques to reduce the size of the documents for storage. Tamino by Software AG and the Dynamic Application Platform of eXcelon are two popular products that offer native XML DBMS capability. Oracle also offers a native XML storage option.

4. **Creating or publishing customized XML documents from preexisting relational databases.** Because there are enormous amounts of data already stored in relational databases, parts of this data may need to be formatted as documents for exchanging or displaying over the Web. This approach would use a separate middleware software layer to handle the conversions needed between the relational data and the extracted XML documents. Section 13.6 discusses this approach, in which data-centric XML documents are extracted from existing databases, in more detail. In particular, we show how tree structured documents can be created from flat relational databases that have

been designed using the ER graph-structured data model. Section 13.6.2 discusses the problem of cycles and how to deal with it.

All of these approaches have received considerable attention. We focus on the fourth approach in Section 13.6, because it gives a good conceptual understanding of the differences between the XML tree data model and the traditional database models based on flat files (relational model) and graph representations (ER model). But first we give an overview of XML query languages in Section 13.5.

13.5 XML Languages

There have been several proposals for XML query languages, and two query language standards have emerged. The first is **XPath**, which provides language constructs for specifying path expressions to identify certain nodes (elements) or attributes within an XML document that match specific patterns. The second is **XQuery**, which is a more general query language. XQuery uses XPath expressions but has additional constructs. We give an overview of each of these languages in this section. Then we discuss some additional languages related to HTML in Section 13.5.3.

13.5.1 XPath: Specifying Path Expressions in XML

An XPath expression generally returns a sequence of items that satisfy a certain pattern as specified by the expression. These items are either values (from leaf nodes) or elements or attributes. The most common type of XPath expression returns a collection of element or attribute nodes that satisfy certain patterns specified in the expression. The names in the XPath expression are node names in the XML document tree that are either tag (element) names or attribute names, possibly with additional **qualifier conditions** to further restrict the nodes that satisfy the pattern. Two main **separators** are used when specifying a path: single slash (/) and double slash (//). A single slash before a tag specifies that the tag must appear as a direct child of the previous (parent) tag, whereas a double slash specifies that the tag can appear as a descendant of the previous tag *at any level*. To refer to an attribute name instead of an element (tag) name, the prefix @ is used before the attribute name. Let us look at some examples of XPath as shown in Figure 13.6.

The first XPath expression in Figure 13.6 returns the company root node and all its descendant nodes, which means that it returns the whole XML document. We should note that it is customary to include the file name in the XPath query. This allows us to specify any local file name or even any path name that specifies a file on the Web. For example, if the COMPANY XML document is stored at the location

www.company.com/info.XML

then the first XPath expression in Figure 13.6 can be written as

doc(www.company.com/info.XML)/company

This prefix would also be included in the other examples of XPath expressions.

Figure 13.6
Some examples of
XPath expressions
on XML documents
that follow the XML
schema file *company*
in Figure 13.5.

1. /company

2. /company/department

3. //employee [employeeSalary gt 70000]/employeeName

4. /company/employee [employeeSalary gt 70000]/employeeName

5. /company/project/projectWorker [hours ge 20.0]

The second example in Figure 13.6 returns all department nodes (elements) and their descendant subtrees. Note that the nodes (elements) in an XML document are ordered, so the XPath result that returns multiple nodes will do so in the same order in which the nodes are ordered in the document tree.

The third XPath expression in Figure 13.6 illustrates the use of //, which is convenient to use if we do not know the full path name we are searching for, but we do know the name of some tags of interest within the XML document. This is particularly useful for schemaless XML documents or for documents with many nested levels of nodes.[6]

The expression returns all employeeName nodes that are direct children of an employee node, such that the employee node has another child element employeeSalary whose value is greater than 70000. This illustrates the use of qualifier conditions, which restrict the nodes selected by the XPath expression to those that satisfy the condition. XPath has a number of comparison operations for use in qualifier conditions, including standard arithmetic, string, and set comparison operations.

The fourth XPath expression in Figure 13.6 should return the same result as the previous one, except that we specified the full path name in this example. The fifth expression in Figure 13.6 returns all projectWorker nodes and their descendant nodes that are children under a path /company/project and have a child node, hours, with a value greater than 20.0 hours.

When we need to include attributes in an XPath expression, the attribute name is prefixed by the @ symbol to distinguish it from element (tag) names. It is also possible to use the **wildcard** symbol *, which stands for any element, as in the following example, which retrieves all elements that are child elements of the root, regardless of their element type. When wildcards are used, the result can be a sequence of different types of elements.

/company/*

The examples above illustrate simple XPath expressions, where we can only move down in the tree structure from a given node. A more general model for path expressions has been proposed. In this model, it is possible to move in multiple directions from the current node in the path expression. These are known as the

---

[6]We use the terms *node*, *tag*, and *element* interchangeably here.

axes of an XPath expression. Our examples above used only *three of these axes*: child of the current node (/), descendent or self at any level of the current node (//), and attribute of the current node (@). Other axes include parent, ancestor (at any level), previous sibling (a node at same level to the left), and next sibling (a node at the same level to the right). These axes allow for more complex path expressions.

The main restriction of XPath path expressions is that the path that specifies the pattern also specifies the items to be retrieved. Hence, it is difficult to specify certain conditions on the pattern while separately specifying which result items should be retrieved. The XQuery language separates these two concerns and provides more powerful constructs for specifying queries.

13.5.2 XQuery: Specifying Queries in XML

XPath allows us to write expressions that select items from a tree-structured XML document. XQuery permits the specification of more general queries on one or more XML documents. The typical form of a query in XQuery is known as a **FLWOR expression**, which stands for the five main clauses of XQuery and has the following form:

```
FOR <variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
ORDER BY <ordering specifications>
RETURN <query result specification>
```

There can be zero or more instances of the FOR clause, as well as of the LET clause in a single XQuery. The WHERE and ORDER BY clauses are optional but can appear at most once, and the RETURN clause must appear exactly once. Let us illustrate these clauses with the following simple example of an XQuery.

```
LET $d : = doc(www.company.com/info.xml)
FOR $x IN $d/company/project[projectNumber = 5]/projectWorker,
       $y IN $d/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
ORDER BY $x/hours
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName,
                $x/hours </res>
```

1. Variables are prefixed with the $ sign. In the above example, $d, $x, and $y are variables. The LET clause assigns a variable to a particular expression for the rest of the query. In this example, $d is assigned to the document file name. It is possible to have a query that refers to multiple documents by assigning multiple variables in this way.

2. The FOR clause assigns a variable to range over each of the individual elements in a sequence. In our example, the sequences are specified by path expressions. The $x variable ranges over elements that satisfy the path expression $d/company/project[projectNumber = 5]/projectWorker. The $y variable

ranges over elements that satisfy the path expression $d/company/employee. Hence, $x ranges over projectWorker elements for workers who work in project 5, whereas $y ranges over employee elements.

3. The WHERE clause specifies additional conditions on the selection of items. In this example, the first condition selects only those projectWorker elements that satisfy the condition (hours gt 20.0). The second condition specifies a join condition that combines an employee with a projectWorker only if they have the same ssn value.

4. The ORDER BY clause specifies that the result elements will be ordered by the value of the hours per week they work on the project in ascending value of hours.

5. Finally, the RETURN clause specifies which elements or attributes should be retrieved from the items that satisfy the query conditions. In this example, it will return a sequence of elements each containing <firstName, lastName, hours> for employees who work more that 20 hours per week on project number 5.

Figure 13.7 includes some additional examples of queries in XQuery that can be specified on an XML instance documents that follow the XML schema document in Figure 13.5. The first query retrieves the first and last names of employees who earn more than $70,000. The variable $x is bound to each employeeName element that is a child of an employee element, but only for employee elements that satisfy the qualifier that their employeeSalary value is greater than $70,000. The result retrieves the firstName and lastName child elements of the selected employeeName elements. The second query is an alternative way of retrieving the same elements retrieved by the first query.

The third query illustrates how a join operation can be performed by using more than one variable. Here, the $x variable is bound to each projectWorker element that is a child of project number 5, whereas the $y variable is bound to each employee element. The join condition matches ssn values in order to retrieve the employee names. Notice that this is an alternative way of specifying the same query in our earlier example, but without the LET clause.

XQuery has very powerful constructs to specify complex queries. In particular, it can specify universal and existential quantifiers in the conditions of a query, aggregate functions, ordering of query results, selection based on position in a sequence, and even conditional branching. Hence, in some ways, it qualifies as a full-fledged programming language.

This concludes our brief introduction to XQuery. The interested reader is referred to www.w3.org, which contains documents describing the latest standards related to XML and XQuery. The next section briefly discusses some additional languages and protocols related to XML.

13.5.3 Other Languages and Protocols Related to XML

There are several other languages and protocols related to XML technology. The long-term goal of these and other languages and protocols is to provide the

1. FOR $x IN
   doc(www.company.com/info.xml)
   //employee [employeeSalary gt 70000]/employeeName
   RETURN <res> $x/firstName, $x/lastName </res>

2. FOR $x IN
   doc(www.company.com/info.xml)/company/employee
   WHERE $x/employeeSalary gt 70000
   RETURN <res> $x/employeeName/firstName, $x/employeeName/lastName </res>

3. FOR $x IN
   doc(www.company.com/info.xml)/company/project[projectNumber=5]/projectWorker,
   $y IN doc(www.company.com/info.xml)/company/employee
   WHERE $x/hours gt 20.0 AND $y.ssn=$x.ssn
   RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>

technology for realization of the Semantic Web, where all information in the Web can be intelligently located and processed.

■ The Extensible Stylesheet Language (XSL) can be used to define how a document should be rendered for display by a Web browser.

■ The Extensible Stylesheet Language for Transformations (XSLT) can be used to transform one structure into a different structure. Hence, it can convert documents from one form to another.

■ The Web Services Description Language (WSDL) allows for the description of Web Services in XML. This makes the Web Service available to users and programs over the Web.

■ The Simple Object Access Protocol (SOAP) is a platform-independent and programming language-independent protocol for messaging and remote procedure calls.

■ The Resource Description Framework (RDF) provides languages and tools for exchanging and processing of meta-data (schema) descriptions and specifications over the Web.

13.6 Extracting XML Documents from Relational Databases

13.6.1 Creating Hierarchical XML Views over Flat or Graph-Based Data

This section discusses the representational issues that arise when converting data from a database system into XML documents. As we have discussed, XML uses a hierarchical (tree) model to represent documents. The database systems with the most widespread use follow the flat relational data model. When we add referential

integrity constraints, a relational schema can be considered to be a graph structure (for example, see Figure 3.7). Similarly, the ER model represents data using graph-like structures (for example, see Figure 7.2). We saw in Chapter 9 that there are straightforward mappings between the ER and relational models, so we can conceptually represent a relational database schema using the corresponding ER schema. Although we will use the ER model in our discussion and examples to clarify the conceptual differences between tree and graph models, the same issues apply to converting relational data to XML.

We will use the simplified UNIVERSITY ER schema shown in Figure 13.8 to illustrate our discussion. Suppose that an application needs to extract XML documents for student, course, and grade information from the UNIVERSITY database. The data needed for these documents is contained in the database attributes of the entity types COURSE, SECTION, and STUDENT from Figure 13.8, and the relationships S-S and C-S between them. In general, most documents extracted

Figure 13.8
An ER schema diagram for a simplified UNIVERSITY database.

Figure 13.9
Subset of the UNIVERSITY database schema needed for XML document extraction.

from a database will only use a subset of the attributes, entity types, and relationships in the database. In this example, the subset of the database that is needed is shown in Figure 13.9.

At least three possible document hierarchies can be extracted from the database subset in Figure 13.9. First, we can choose COURSE as the root, as illustrated in Figure 13.10. Here, each course entity has the set of its sections as subelements, and each section has its students as subelements. We can see one consequence of modeling the information in a hierarchical tree structure. If a student has taken multiple sections, that student's information will appear multiple times in the document—once under each section. A possible simplified XML schema for this view is shown in Figure 13.11. The Grade database attribute in the S-S relationship is migrated to the STUDENT element. This is because STUDENT becomes a child of SECTION in this hierarchy, so each STUDENT element under a specific SECTION element can have a

Figure 13.10
Hierarchical (tree) view with COURSE as the root.

```
<xsd:element name="root">
    <xsd:sequence>
    <xsd:element name="course" minOccurs="0" maxOccurs="unbounded">
        <xsd:sequence>
            <xsd:element name="cname" type="xsd:string" />
            <xsd:element name="cnumber" type="xsd:unsignedInt" />
            <xsd:element name="section" minOccurs="0" maxOccurs="unbounded">
                <xsd:sequence>
                    <xsd:element name="secnumber" type="xsd:unsignedInt" />
                    <xsd:element name="year" type="xsd:string" />
                    <xsd:element name="quarter" type="xsd:string" />
                    <xsd:element name="student" minOccurs="0" maxOccurs="unbounded">
                        <xsd:sequence>
                            <xsd:element name="ssn" type="xsd:string" />
                            <xsd:element name="sname" type="xsd:string" />
                            <xsd:element name="class" type="xsd:string" />
                            <xsd:element name="grade" type="xsd:string" />
                        </xsd:sequence>
                    </xsd:element>
                </xsd:sequence>
            </xsd:element>
        </xsd:sequence>
    </xsd:element>
    </xsd:sequence>
</xsd:element>
```
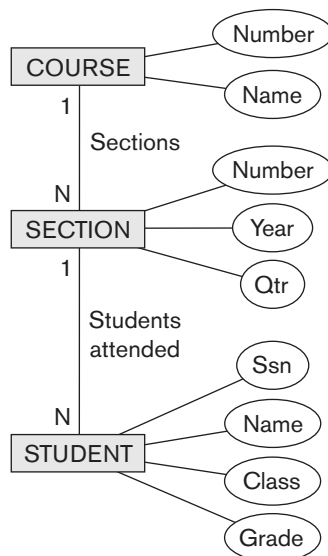
Figure 13.11
XML schema document with *course* as the root.

specific grade in that section. In this document hierarchy, a student taking more than one section will have several replicas, one under each section, and each replica will have the specific grade given in that particular section.

In the second hierarchical document view, we can choose STUDENT as root (Figure 13.12). In this hierarchical view, each student has a set of sections as its child elements, and each section is related to one course as its child, because the relationship between SECTION and COURSE is N:1. Thus, we can merge the COURSE and SECTION elements in this view, as shown in Figure 13.12. In addition, the GRADE database attribute can be migrated to the SECTION element. In this hierarchy, the combined COURSE/SECTION information is replicated under each student who completed the section. A possible simplified XML schema for this view is shown in Figure 13.13.

The third possible way is to choose SECTION as the root, as shown in Figure 13.14. Similar to the second hierarchical view, the COURSE information can be merged into the SECTION element. The GRADE database attribute can be migrated to the

Figure 13.12
Hierarchical (tree) view with STUDENT as the root.

Figure 13.13
XML schema document with *student* as the root.

```
<xsd:element name="root">
<xsd:sequence>
<xsd:element name="student" minOccurs="0" maxOccurs="unbounded">
    <xsd:sequence>
        <xsd:element name="ssn" type="xsd:string" />
        <xsd:element name="sname" type="xsd:string" />
        <xsd:element name="class" type="xsd:string" />
        <xsd:element name="section" minOccurs="0" maxOccurs="unbounded">
            <xsd:sequence>
                <xsd:element name="secnumber" type="xsd:unsignedInt" />
                <xsd:element name="year" type="xsd:string" />
                <xsd:element name="quarter" type="xsd:string" />
                <xsd:element name="cnumber" type="xsd:unsignedInt" />
                <xsd:element name="cname" type="xsd:string" />
                <xsd:element name="grade" type="xsd:string" />
            </xsd:sequence>
        </xsd:element>
    </xsd:sequence>
</xsd:element>
</xsd:sequence>
</xsd:element>
```

Figure 13.14
Hierarchical (tree) view with SECTION as the root.

STUDENT element. As we can see, even in this simple example, there can be numerous hierarchical document views, each corresponding to a different root and a different XML document structure.

13.6.2 Breaking Cycles to Convert Graphs into Trees

In the previous examples, the subset of the database of interest had no cycles. It is possible to have a more complex subset with one or more cycles, indicating multiple relationships among the entities. In this case, it is more difficult to decide how to create the document hierarchies. Additional duplication of entities may be needed to represent the multiple relationships. We will illustrate this with an example using the ER schema in Figure 13.8.

Suppose that we need the information in all the entity types and relationships in Figure 13.8 for a particular XML document, with STUDENT as the root element. Figure 13.15 illustrates how a possible hierarchical tree structure can be created for this document. First, we get a lattice with STUDENT as the root, as shown in Figure 13.15(a). This is not a tree structure because of the cycles. One way to break the cycles is to replicate the entity types involved in the cycles. First, we replicate INSTRUCTOR as shown in Figure 13.15(b), calling the replica to the right INSTRUCTOR1. The INSTRUCTOR replica on the left represents the relationship between instructors and the sections they teach, whereas the INSTRUCTOR1 replica on the right represents the relationship between instructors and the department each works in. After this, we still have the cycle involving COURSE, so we can replicate COURSE in a similar manner, leading to the hierarchy shown in Figure 13.15(c). The COURSE1 replica to the left represents the relationship between courses and their sections, whereas the COURSE replica to the right represents the relationship between courses and the department that offers each course.

In Figure 13.15(c), we have converted the initial graph to a hierarchy. We can do further merging if desired (as in our previous example) before creating the final hierarchy and the corresponding XML schema structure.

Figure 13.15
Converting a graph with cycles into a hierarchical (tree) structure.

13.6.3 Other Steps for Extracting XML Documents from Databases

In addition to creating the appropriate XML hierarchy and corresponding XML schema document, several other steps are needed to extract a particular XML document from a database:

1. It is necessary to create the correct query in SQL to extract the desired information for the XML document.

2. Once the query is executed, its result must be restructured from the flat relational form to the XML tree structure.

3. The query can be customized to select either a single object or multiple objects into the document. For example, in the view in Figure 13.13, the query can select a single student entity and create a document corresponding to that single student, or it may select several—or even all—of the students and create a document with multiple students.

13.7 XML/SQL: SQL Functions for Creating XML Data

In this section, we discuss some of the functions that have been added to the recent versions of the SQL standard for the purpose of generating XML data from relational databases. These functions can be used to format the results of queries into XML elements and documents, and to specify the roots of an XML hierarchy so that nested hierarchical data can be created from flat relational data. First we list and briefly describe some of the functions that were added to SQL; then we show a few examples.

We discuss the following functions:

1. XMLELEMENT: This is used to specify a tag (element) name that will appear in the XML result. It can specify a tag name for a complex element or for an individual column.
2. XMLFOREST: If several tags (elements) are needed in the XML result, this function can create multiple element names in a simpler manner than XMLELEMENT. The column names can be listed directly, separated by commas, with or without renaming. If a column name is not renamed, it will be used as the element (tag) name.
3. XMLAGG: This can group together (or aggregate) several elements so they can be placed under a parent element as a collection of subelements.
4. XMLROOT: This allows the selected elements to be formatted as an XML document with a single root element.
5. XMLATTRIBUTES: This allows the creation of attributes for the elements of the XML result.

We now illustrate these functions with a few SQL/XML examples that refer to the EMPLOYEE table from Figures 5.5 and 5.6. The first example X1 shows how to create an XML element that contains the EMPLOYEE lastname for the employee whose ssn is "123456789":

```
X1:  SELECT    XMLELEMENT (NAME "lastname", E.LName)
     FROM      EMPLOYEE E
     WHERE     E.Ssn = "123456789" ;
```

The SQL keyword NAME specifies the XML element (tag) name. The result on the data shown in Figure 5.6 would be:

```
<lastname>Smith</lastname>
```

If we want to retrieve multiple columns for a single row, we can use multiple listings of XMLELEMENT within the parent element, but a simpler way would be to use XMLFOREST, which allows the specification of multiple columns without repeating the keyword XMLELEMENT multiple times. This is shown as X2:

```
X2:  SELECT    XMLELEMENT (NAME "employee",
                   XMLFOREST (
                         E.Lname AS "ln",
                         E.Fname AS "fn",
                         E.Salary AS "sal" ) )
     FROM      EMPLOYEE AS E
     WHERE     E.Ssn = "123456789" ;
```

The result of X2 on the data shown in Figure 5.6 would be:

```
<employee><ln>Smith</ln><fn>John</fn><sal>30000</sal></employee>
```

Suppose we want to create XML data that has the last name, first name, and salary of the employees who work in department 4, and format it as an XML

document with the root tag "dept4emps". Then we can write the SQL/XML query X3:

X3: SELECT   **XMLROOT** (
                        XMLELEMENT (**NAME** "dept4emps",

                        XMLAGG (
                            XMLELEMENT (**NAME** "emp"
                            XMLFOREST (Lname, Fname, Salary)
                            ORDER BY Lname)))

   FROM   EMPLOYEE
   WHERE   Dno = 4 ;

The XMLROOT function creates a single root element, so the XML data would be a well-formed document (a tree with a single root). The result of X3 on the data shown in Figure 5.6 would be:

```
<dept4emps>
<emp><Lname>Jabbar</Lname><Fname>Ahmad</Fname><Salary>25000
   </Salary></emp>
<emp><Lname>Wallace</Lname><Fname>Jennifer
   </Fname><Salary>43000</Salary></emp>
<emp><Lname>Zelaya</Lname><Fname>Alicia</Fname><Salary>25000
   </Salary></emp>
</dept4emps>
```

These examples give a flavor of how the SQL standard has been extended to allow users to format query results as XML data.

13.8 Summary

This chapter provided an overview of the XML standard for representing and exchanging data over the Internet. First we discussed some of the differences between various types of data, classifying three main types: structured, semistructured, and unstructured. Structured data is stored in traditional databases. Semistructured data mixes data types names and data values, but the data does not all have to follow a fixed predefined structure. Unstructured data refers to information displayed on the Web, specified via HTML, where information on the types of data items is missing. We described the XML standard and its tree-structured (hierarchical) data model, and we discussed XML documents and the languages for specifying the structure of these documents, namely, XML DTD (Document Type Definition) and XML schema. We gave an overview of the various approaches for storing XML documents, whether in their native (text) format, in a compressed form, or in relational and other types of databases. We gave an overview of the XPath and XQuery languages proposed for querying XML data, and we discussed the mapping issues that arise when it is necessary to convert data stored in traditional relational databases into XML documents. Finally, we discussed SQL/XML, which provides SQL with additional functionality to format SQL query results as XML data.

Review Questions

13.1. What are the differences between structured, semistructured, and unstructured data?

13.2. Under which of the categories mentioned in Question 13.1 do XML documents fall? What about self-describing data?

13.3. What are the differences between the use of tags in XML versus HTML?

13.4. What is the difference between data-centric and document-centric XML documents?

13.5. What is the difference between attributes and elements in XML? List some of the important attributes used to specify elements in XML schema.

13.6. What is the difference between XML schema and XML DTD?

Exercises

13.7. Create part of an XML instance document to correspond to the data stored in the relational database shown in Figure 5.6 such that the XML document conforms to the XML schema document in Figure 13.5.

13.8. Create XML schema documents and XML DTDs to correspond to the hierarchies shown in Figures 13.14 and 13.15(c).

13.9. Consider the LIBRARY relational database schema in Figure 6.6. Create an XML schema document that corresponds to this database schema.

13.10. Specify the following views as queries in XQuery on the *company* XML schema shown in Figure 13.5.

    a. A view that has the department name, manager name, and manager salary for every department

    b. A view that has the employee name, supervisor name, and employee salary for each employee who works in the Research department

    c. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project

    d. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project with more than one employee working on it

Selected Bibliography

There are so many articles and books on various aspects of XML that it would be impossible to make even a modest list. We will mention one book: Chaudhri, Rashid, and Zicari, editors (2003). This book discusses various aspects of XML and contains a list of references to XML research and practice.

part **6**

Database Design Theory and Normalization

This page intentionally left blank

Basics of Functional Dependencies and Normalization for Relational Databases

I n Chapters 5 through 8, we presented various aspects of the relational model and the languages associated with it. Each *relation schema* consists of a number of attributes, and the *relational database schema* consists of a number of relation schemas. So far, we have assumed that attributes are grouped to form a relation schema by using the common sense of the database designer or by mapping a database schema design from a conceptual data model such as the ER or enhanced-ER (EER) data model. These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations when the mapping procedures discussed in Chapter 9 are followed. However, we still need some formal way of analyzing why one grouping of attributes into a relation schema may be better than another. While discussing database design in Chapters 3, 4, and 9, we did not develop any measure of appropriateness or *goodness* to measure the quality of the design, other than the intuition of the designer. In this chapter we discuss some of the theory that has been developed with the goal of evaluating relational schemas for design quality—that is, to measure formally why one set of groupings of attributes into relation schemas is better than another.

There are two levels at which we can discuss the *goodness* of relation schemas. The first is the **logical** (or **conceptual**) **level**—how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their queries correctly. The second is the **implementation** (or **physical storage**) **level**—how the tuples in a base relation are stored and updated.

This level applies only to schemas of base relations—which will be physically stored as files—whereas at the logical level we are interested in schemas of both base relations and views (virtual relations). The relational database design theory developed in this chapter applies mainly to *base relations,* although some criteria of appropriateness also apply to views, as shown in Section 14.1.

As with many design problems, database design may be performed using two approaches: bottom-up or top-down. A **bottom-up design methodology** (also called *design by synthesis*) considers the basic relationships *among individual attributes* as the starting point and uses those to construct relation schemas. This approach is not very popular in practice[1] because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes. In contrast, a **top-down design methodology** (also called *design by analysis*) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met. The theory described in this chapter is applicable primarily to the top-down design approach, and as such is more appropriate when performing design of databases by analysis and decomposition of sets of attributes that appear together in files, in reports, and on forms in real-life situations.

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are *information preservation* and *minimum redundancy*. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships, which are described using a model such as the EER model. Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.

We start this chapter by informally discussing some criteria for good and bad relation schemas in Section 14.1. In Section 14.2, we define the concept of *functional dependency,* a formal constraint among attributes that is the main tool for formally measuring the appropriateness of attribute groupings into relation schemas. In Section 14.3, we discuss normal forms and the process of normalization using functional dependencies. Successive normal forms are defined to meet a set of desirable constraints expressed using primary keys and functional dependencies. The normalization procedure consists of applying a series of tests to relations to meet these increasingly stringent requirements and decompose the relations when necessary. In Section 14.4, we discuss more general definitions of normal forms that can be directly

---

[1]An exception in which this approach is used in practice is based on a model called the *binary relational model*. An example is the NIAM methodology (Verheijen and VanBekkum, 1982).

applied to any given design and do not require step-by-step analysis and normalization. Sections 14.5 to 14.7 discuss further normal forms up to the fifth normal form. In Section 14.6 we introduce the multivalued dependency (MVD), followed by the join dependency (JD) in Section 14.7. Section 14.8 summarizes the chapter.

Chapter 15 continues the development of the theory related to the design of good relational schemas. We discuss desirable properties of relational decomposition—nonadditive join property and functional dependency preservation property. A general algorithm that tests whether or not a decomposition has the nonadditive (or *lossless*) join property (Algorithm 15.3 is also presented). We then discuss properties of functional dependencies and the concept of a minimal cover of dependencies. We consider the bottom-up approach to database design consisting of a set of algorithms to design relations in a desired normal form. These algorithms assume as input a given set of functional dependencies and achieve a relational design in a target normal form while adhering to the above desirable properties. In Chapter 15 we also define additional types of dependencies that further enhance the evaluation of the *goodness* of relation schemas.

If Chapter 15 is not covered in a course, we recommend a quick introduction to the desirable properties of decomposition from Section 15.2. and the importance of the non-additive join property during decomposition.

14.1 Informal Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, we discuss four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another, as we will see.

14.1.1 Imparting Clear Semantics to Attributes in Relations

Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them. The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. In Chapter 5 we discussed how a relation can be interpreted as a set of facts. If the conceptual design described in Chapters 3 and 4 is done carefully and the mapping procedure in Chapter 9 is followed systematically, the relational schema design should have a clear meaning.

In general, the easier it is to explain the semantics of the relation—or in other words, what a relation exactly means and stands for—the better the relation schema design will be. To illustrate this, consider Figure 14.1, a simplified version of the COMPANY relational database schema in Figure 5.5, and Figure 14.2, which presents an example of populated relation states of this schema. The meaning of the EMPLOYEE relation schema is simple: Each tuple represents an employee, with values for the employee's name (Ename), Social Security number (Ssn), birth date (Bdate), and address (Address), and the number of the department that the employee works for (Dnumber). The Dnumber attribute is a foreign key that represents an *implicit relationship* between EMPLOYEE and DEPARTMENT. The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute Dmgr_ssn of DEPARTMENT relates a department to the employee who is its manager, whereas Dnum of PROJECT relates a project to its controlling department; both are foreign key attributes. The ease with which the meaning of a relation's attributes can be explained is an *informal measure* of how well the relation is designed.

Figure 14.1

A simplified COMPANY relational database schema.

EMPLOYEE   F.K.

Ename	Ssn	Bdate	Address	Dnumber

            P.K.

DEPARTMENT   F.K.

Dname	Dnumber	Dmgr_ssn

              P.K.

DEPT_LOCATIONS
        F.K.

Dnumber	Dlocation

            P.K.

PROJECT   F.K.

Pname	Pnumber	Plocation	Dnum

              P.K.

WORKS_ON
        F.K.   F.K.

Ssn	Pnumber	Hours

            P.K.

Figure 14.2
Sample database state for the relational database schema in Figure 14.1.

EMPLOYEE

Ename	Ssn	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Ssn	Pnumber	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	Null

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

The semantics of the other two relation schemas in Figure 14.1 are slightly more complex. Each tuple in DEPT_LOCATIONS gives a department number (Dnumber) and *one of* the locations of the department (Dlocation). Each tuple in WORKS_ON gives an employee Social Security number (Ssn), the project number of *one of* the projects that the employee works on (Pnumber), and the number of hours per week that the employee works on that project (Hours). However, both schemas have a well-defined and unambiguous interpretation. The schema DEPT_LOCATIONS represents a multivalued attribute of DEPARTMENT, whereas WORKS_ON represents an M:N relationship between EMPLOYEE and PROJECT. Hence, all the relation schemas in Figure 14.1 may be considered as easy to explain and therefore good from the standpoint of having clear semantics. We can thus formulate the following informal design guideline.

Guideline 1. Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Violating Guideline 1. The relation schemas in Figures 14.3(a) and 14.3(b) also have clear semantics. (The reader should ignore the lines under the relations for now; they are used to illustrate functional dependency notation, discussed in Section 14.2.) A tuple in the EMP_DEPT relation schema in Figure 14.3(a) represents a single employee but includes, along with the Dnumber (the identifier for the department he/she works for), additional information—namely, the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssn) of the department manager. For the EMP_PROJ relation in Figure 14.3(b), each tuple relates an employee to a project but also includes

Figure 14.3
Two relation schemas suffering from update anomalies.
(a) EMP_DEPT and
(b) EMP_PROJ.

the employee name (Ename), project name (Pname), and project location (Plocation). Although there is nothing wrong logically with these two relations, they violate Guideline 1 by mixing attributes from distinct real-world entities: EMP_DEPT mixes attributes of employees and departments, and EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship. Hence, they fare poorly against the above measure of design quality. They may be used as views, but they cause problems when used as base relations, as we discuss in the following section.

14.1.2 Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 14.2 with that for an EMP_DEPT base relation in Figure 14.4, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for *every employee who works for that department*. In contrast, each department's information appears only once in the DEPARTMENT relation in Figure 14.2. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation (see Figure 14.4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.[2]

Insertion Anomalies. Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

- To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT. In the design of Figure 14.2, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation.

- It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the

---

[2]These anomalies were identified by Codd (1972a) to justify the need for normalization of relations, as we shall discuss in Section 15.3.

Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Redundancy   Redundancy

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation
123456789	1	32.5	Smith, John B.	ProductX	Bellaire
123456789	2	7.5	Smith, John B.	ProductY	Sugarland
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston
888665555	20	Null	Borg, James E.	Reorganization	Houston

Figure 14.4
Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations
in Figure 14.2. These may be stored as base relations for performance reasons.

attributes for employee. This violates the entity integrity for EMP_DEPT
because its primary key Ssn cannot be null. Moreover, when the first
employee is assigned to that department, we do not need this tuple with
NULL values anymore. This problem does not occur in the design of Fig-
ure 14.2 because a department is entered in the DEPARTMENT relation whether
or not any employees work for it, and whenever an employee is assigned to
that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies. The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost inadvertently from the database. This problem does not occur in the database of Figure 14.2 because DEPARTMENT tuples are stored separately.

Modification Anomalies. In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.[3]

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided; hence, we can state the next guideline as follows.

Guideline 2. Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present,[4] note them clearly and make sure that the programs that update the database will operate correctly.

The second guideline is consistent with and, in a way, a restatement of the first guideline. We can also see the need for a more formal approach to evaluating whether a design meets these guidelines. Sections 14.2 through 14.4 provide these needed formal concepts. It is important to note that these guidelines may sometimes *have to be violated* in order to *improve the performance* of certain queries. If EMP_DEPT is used as a stored relation (known otherwise as a *materialized view*) in addition to the base relations of EMPLOYEE and DEPARTMENT, the anomalies in EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates). This way, whenever the base relation is updated, we do not end up with inconsistencies. In general, it is advisable to use anomaly-free base relations and to specify views that include the joins for placing together the attributes frequently referenced in important queries.

14.1.3 NULL Values in Tuples

In some schema designs we may group many attributes together into a "fat" relation. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with

---

[3]This is not as serious as the other problems, because all tuples can be updated by a single SQL query.

[4]Other application considerations may dictate and make certain anomalies unavoidable. For example, the EMP_DEPT relation may correspond to a query or a report that is frequently required.

specifying JOIN operations at the logical level.[5] Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.[6] Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
- The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
- The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we state another guideline.

Guideline 3. As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Using space efficiently and avoiding joins with NULL values are the two overriding criteria that determine whether to include the columns that may have NULLs in a relation or to have a separate relation for those columns (with the appropriate key columns). For example, if only 15% of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created to include tuples for only the employees with individual offices.

14.1.4 Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 14.5(a), which can be used instead of the single EMP_PROJ relation in Figure 14.3(b). A tuple in EMP_LOCS means that the employee whose name is Ename works on *at least one* project located at Plocation. A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works the given Hours per week on the project whose name, number, and location are Pname, Pnumber, and Plocation. Figure 14.5(b) shows relation states of EMP_LOCS and EMP_PROJ1 corresponding to the EMP_PROJ relation in Figure 14.4, which are obtained by applying the appropriate PROJECT (π) operations to EMP_PROJ.

---

[5]This is because inner and outer joins produce different results when NULLs are involved in joins. The users must thus be aware of the different meanings of the various types of joins. Although this is reasonable for sophisticated users, it may be difficult for others.

[6]In Section 5.5.1 we presented comparisons involving NULL values where the outcome (in three-valued logic) is TRUE, FALSE, and UNKNOWN.

(a)

EMP_LOCS

Ename	Plocation

P.K.

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation

P.K.

Figure 14.5
Particularly poor design for the EMP_PROJ relation in
Figure 14.3(b). (a) The two relation schemas EMP_LOCS
and EMP_PROJ1. (b) The result of projecting the
extension of EMP_PROJ from Figure 14.4 onto the
relations EMP_LOCS and EMP_PROJ1.

(b)

EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of
EMP_PROJ. This produces a particularly bad schema design because we cannot
recover the information that was originally in EMP_PROJ from EMP_PROJ1 and
EMP_LOCS. If we attempt a NATURAL JOIN operation on EMP_PROJ1 and
EMP_LOCS, the result produces many more tuples than the original set of tuples
in EMP_PROJ. In Figure 14.6, the result of applying the join to only the tuples for
employee with Ssn = "123456789" is shown (to reduce the size of the resulting rela-
tion). Additional tuples that were not in EMP_PROJ are called **spurious tuples**
because they represent spurious information that is not valid. The spurious
tuples are marked by asterisks (*) in Figure 14.6. It is left to the reader to complete
the result of NATURAL JOIN operation on the EMP_PROJ1 and EMP_LOCS
tables in their entirety and to mark the spurious tuples in this result.

	Ssn	Pnumber	Hours	Pname	Plocation	Ename
	123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
	123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
	666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	ProductZ	Houston	Wong, Franklin T.
*	453453453	1	20.0	ProductX	Bellaire	Smith, John B.
	453453453	1	20.0	ProductX	Bellaire	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Smith, John B.
	453453453	2	20.0	ProductY	Sugarland	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	2	10.0	ProductY	Sugarland	Smith, John B.
*	333445555	2	10.0	ProductY	Sugarland	English, Joyce A.
	333445555	2	10.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	3	10.0	ProductZ	Houston	Narayan, Ramesh K.
	333445555	3	10.0	ProductZ	Houston	Wong, Franklin T.
	333445555	10	10.0	Computerization	Stafford	Wong, Franklin T.
*	333445555	20	10.0	Reorganization	Houston	Narayan, Ramesh K.
	333445555	20	10.0	Reorganization	Houston	Wong, Franklin T.

*
*
*

Figure 14.6
Result of applying NATURAL JOIN to the tuples in EMP_PROJ1 and EMP_LOCS
of Figure 14.5 just for employee with Ssn = "123456789". Generated spurious
tuples are marked by asterisks.

Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because
when we JOIN them back using NATURAL JOIN, we do not get the correct original
information. This is because in this case Plocation happens to be the attribute that
relates EMP_LOCS and EMP_PROJ1, and Plocation is neither a primary key nor a
foreign key in either EMP_LOCS or EMP_PROJ1. We now informally state another
design guideline.

Guideline 4. Design relation schemas so that they can be joined with equality
conditions on attributes that are appropriately related (primary key, foreign key)
pairs in a way that guarantees that no spurious tuples are generated. Avoid relations
that contain matching attributes that are not (foreign key, primary key) combina-
tions because joining on such attributes may produce spurious tuples.

This informal guideline obviously needs to be stated more formally. In Section 15.2 we discuss a formal condition called the nonadditive (or lossless) join property that guarantees that certain joins do not produce spurious tuples.

14.1.5 Summary and Discussion of Design Guidelines

In Sections 14.1.1 through 14.1.4, we informally discussed situations that lead to problematic relation schemas and we proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

In the rest of this chapter we present formal concepts and theory that may be used to define the *goodness* and *badness* of *individual* relation schemas more precisely. First we discuss functional dependency as a tool for analysis. Then we specify the three normal forms and Boyce-Codd normal form (BCNF) for relation schemas as the established and accepted standards of quality in relational design. The strategy for achieving a good design is to decompose a badly designed relation appropriately to achieve higher normal forms. We also briefly introduce additional normal forms that deal with additional dependencies. In Chapter 15, we discuss the properties of decomposition in detail and provide a variety of algorithms related to functional dependencies, goodness of decomposition, and the bottom-up design of relations by using the functional dependencies as a starting point.

14.2 Functional Dependencies

So far we have dealt with the informal measures of database design. We now introduce a formal tool for analysis of relational schemas that enables us to detect and describe some of the above-mentioned problems in precise terms. The single most important concept in relational schema design theory is that of a functional dependency. In this section we formally define the concept, and in Section 14.3 we see how it can be used to define normal forms for relation schemas.

14.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes A_1, A_2, ... , A_n; let us think of the whole database as being described by a single **universal**

relation schema $R = \{A_1, A_2, \ldots, A_n\}$.[7] We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.[8]

> **Definition.** A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by,* the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X. The abbreviation for functional dependency is **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X-value, they must necessarily agree on their Y-value. Note the following:

- If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance $r(R)$—that is, X is a **candidate key** of R—this implies that $X \rightarrow Y$ for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state $r(R)$ will have the same value of X). If X is a candidate key of R, then $X \rightarrow R$.

- If $X \rightarrow Y$ in R, this does not say whether or not $Y \rightarrow X$ in R.

A functional dependency is a property of the **semantics** or **meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R—that is, how they relate to one another—to specify the functional dependencies that should hold on *all* relation states (extensions) r of R. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R. Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold *at all times.* Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, {State, Driver_license_number} $\rightarrow$ Ssn should normally hold for any adult in the United States and hence should hold whenever these attributes appear in a relation.[9] It is also possible that certain functional

---

[7]This concept of a universal relation is important when we discuss the algorithms for relational database design in Chapter 15.

[8]This assumption implies that every attribute in the database should have a distinct name. In Chapter 5 we prefixed attribute names by relation names to achieve uniqueness whenever attributes in distinct relations had the same name.

[9]Note that there are databases, such as those of credit card agencies or police departments, where this functional dependency may not hold because of fraudulent records resulting from the same driver's license number being used by two or more different individuals.

dependencies may cease to exist in the real world if the relationship changes. For example, the FD Zip_code → Area_code used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

Consider the relation schema EMP_PROJ in Figure 14.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

    a. Ssn → Ename

    b. Pnumber → {Pname, Plocation}

    c. {Ssn, Pnumber} → Hours

These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of Ename,* and so on.

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*. For example, Figure 14.7 shows a *particular state* of the TEACH relation schema. Although at first glance we may think that Text → Course, we cannot confirm this unless we know that it is true *for all possible legal states* of TEACH. It is, however, sufficient to demonstrate *a single counterexample* to disprove a functional dependency. For example, because 'Smith' teaches both 'Data Structures' and 'Database Systems,' we can conclude that Teacher *does not* functionally determine Course.

Given a populated relation, we cannot determine which FDs hold and which do not unless we know the meaning of and the relationships among the attributes. All we can say is that a certain FD *may* exist if it holds in that particular extension. We cannot guarantee its existence until we understand the meaning of the corresponding attributes. We can, however, emphatically state that a certain FD *does not hold* if there are

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Figure 14.7

A relation state of TEACH with a *possible* functional dependency TEXT → COURSE. However, TEACHER → COURSE, TEXT → TEACHER and COURSE → TEXT are ruled out.

Figure 14.8
A relation $R(A, B, C, D)$
with its extension.

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

tuples that show the violation of such an FD. See the illustrative example relation in Figure 14.8. Here, the following FDs *may hold* because the four tuples in the current extension have no violation of these constraints: $B \rightarrow C$; $C \rightarrow B$; $\{A, B\} \rightarrow C$; $\{A, B\} \rightarrow D$; and $\{C, D\} \rightarrow B$. However, the following *do not* hold because we already have violations of them in the given extension: $A \rightarrow B$ (tuples 1 and 2 violate this constraint); $B \rightarrow A$ (tuples 2 and 3 violate this constraint); $D \rightarrow C$ (tuples 3 and 4 violate it).

Figure 14.3 introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, whereas the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

We denote by F the set of functional dependencies that are specified on relation schema R. Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F. Those other dependencies can be *inferred* or *deduced* from the FDs in F. We defer the details of inference rules and properties of functional dependencies to Chapter 15.

14.3 Normal Forms Based on Primary Keys

Having introduced functional dependencies, we are now ready to use them to specify how to use them to develop a formal methodology for testing and improving relation schemas. We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the *normalization process* for relational schema design. Most practical relational design projects take one of the following two approaches:

- Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations.
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports.

Following either of these approaches, it is then useful to evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms using the normalization theory presented in this chapter and the next. We focus in

this section on the first three normal forms for relation schemas and the intuition behind them, and we discuss how they were developed historically. More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key, are deferred to Section 14.4.

We start by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 3 that are needed here. Then we discuss the first normal form (1NF) in Section 14.3.4, and we present the definitions of second normal form (2NF) and third normal form (3NF), which are based on primary keys, in Sections 14.3.5 and 14.3.6, respectively.

14.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis.* Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively; these are briefly discussed in Sections 14.6 and 14.7.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies discussed in Section 14.1.2. It can be considered as a "filtering" or "purification" process to make the design have successively better quality. An unsatisfactory relation schema that does not meet the condition for a normal form—the **normal form test**—is decomposed into smaller relation schemas that contain a subset of the attributes and meet the test that was otherwise not met by the original relation. Thus, the normalization procedure provides database designers with the following:

■ A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes

■ A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

Definition. The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Normal forms, when considered *in isolation* from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each

relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

- The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem discussed in Section 14.1.4 does not occur with respect to the relation schemas created after decomposition
- The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition

The nonadditive join property is extremely critical and **must be achieved at any cost**, whereas the dependency preservation property, although desirable, is sometimes sacrificed, as we discuss in Section 15.2.2. We defer the discussion of the formal concepts and techniques that guarantee the above two properties to Chapter 15.

14.3.2 Practical Use of Normal Forms

Most practical design projects in commercial and governmental environment acquire existing designs of databases from previous designs, from designs in legacy models, or from existing files. They are certainly interested in assuring that the designs are good quality and sustainable over long periods of time. Existing designs are evaluated by applying the tests for normal forms, and normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and 5NF that we discuss in Sections 14.6 and 14.7, the practical utility of these normal forms becomes questionable. The reason is that the constraints on which they are based are rare and hard for the database designers and users to understand or to detect. Designers and users must either already know them or discover them as a part of the business. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers *need not* normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons, such as those discussed at the end of Section 14.1.2. Doing so incurs the corresponding penalties of dealing with the anomalies.

> **Definition. Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

14.3.3 Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let's look again at the definitions of keys of a relation schema from Chapter 3.

> **Definition.** A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey anymore.

The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key $K = \{A_1, A_2, \ldots, A_k\}$ of R, then $K - \{A_i\}$ is not a key of R for any A_i, $1 \leq i \leq k$. In Figure 14.1, {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In Figure 14.1, {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key.

> **Definition.** An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

In Figure 14.1, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF independently attack different types of problems arising from problematic functional dependencies among attributes. However, for historical reasons, it is customary to follow them in that sequence; hence, by definition a 3NF relation *already satisfies* 2NF.
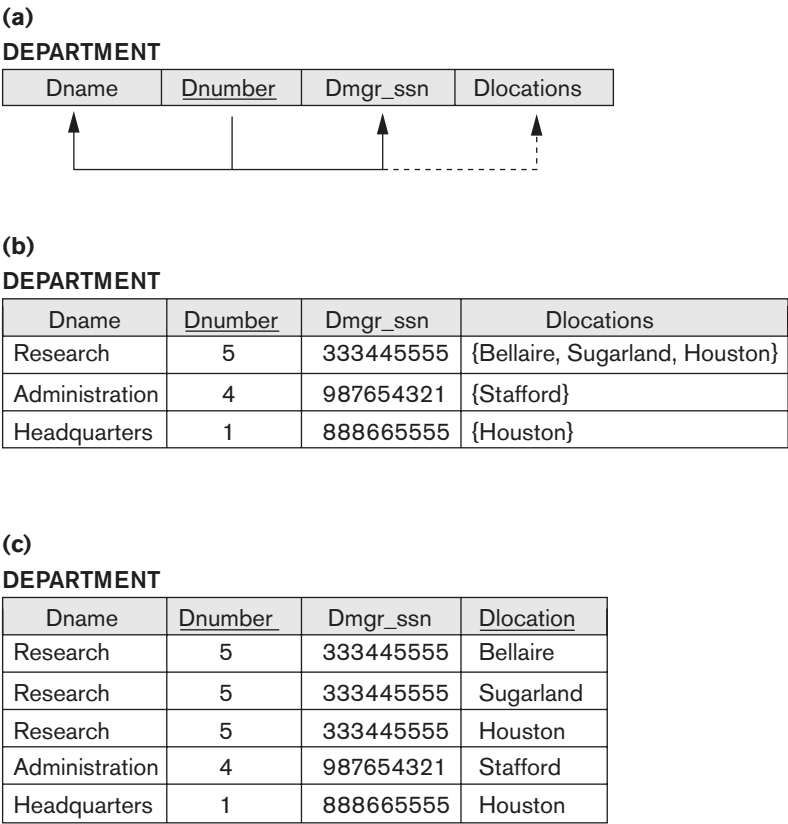
14.3.4 First Normal Form

First normal form (1NF)is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

Consider the DEPARTMENT relation schema shown in Figure 14.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 14.9(a). We assume that each department can have *a number of* locations. The DEPARTMENT schema and a sample relation state are shown in Figure 14.9. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure 14.9(b). There are two ways we can look at the Dlocations attribute:

- ■ The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.

(a)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 14.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

- The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber → Dlocations because each set is considered a single member of the attribute domain.[10]

In either case, the DEPARTMENT relation in Figure 14.9 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 3.1. There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this newly formed relation is the combination {Dnumber, Dlocation}, as shown in Figure 14.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

---

[10]In this case we can consider the domain of Dlocations to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 14.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation and hence is rarely adopted.

3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing *NULL values* if most departments have fewer than three locations. It further introduces spurious semantics about the ordering among the location values; that *ordering* is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: *List the departments that have 'Bellaire' as one of their locations* in this design. For all these reasons, it is best to avoid this alternative.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general; it places no maximum limit on the number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.
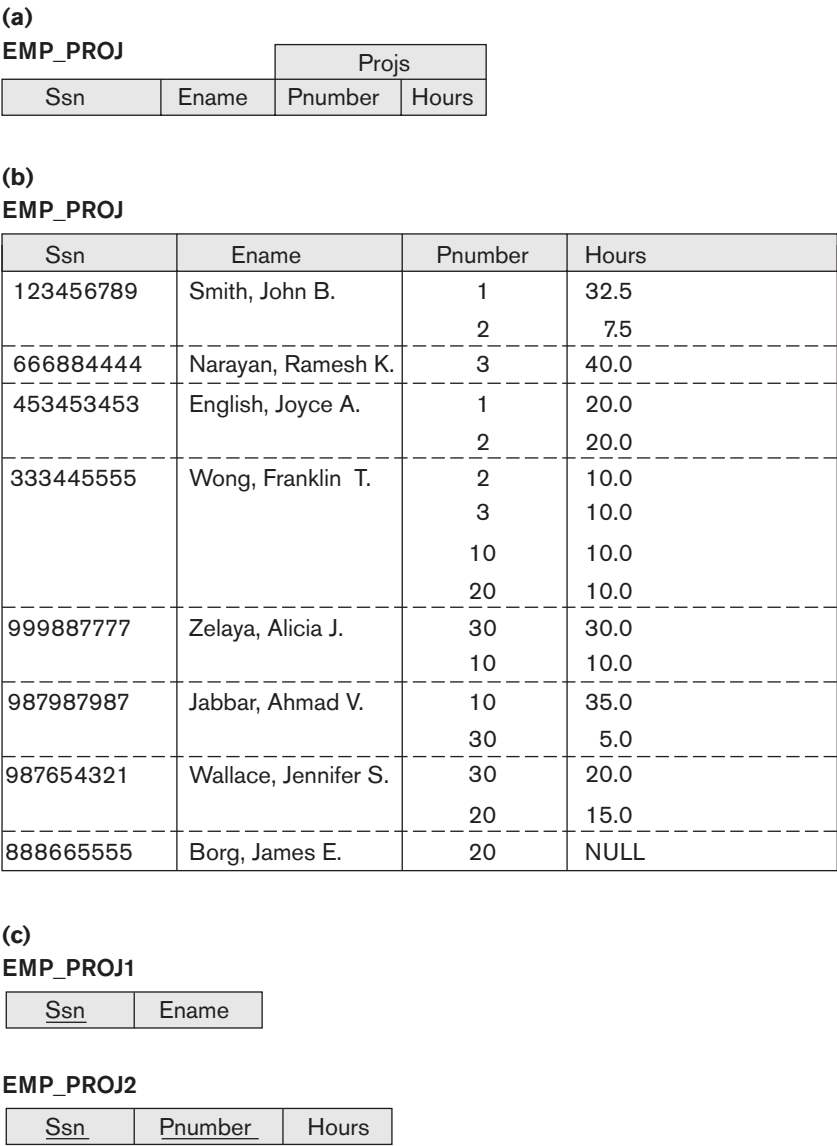
First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it.* Figure 14.10 shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

    EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)})

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Interestingly, recent trends for supporting complex objects (see Chapter 12) and XML data (see Chapter 13) attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that Ssn is the primary key of the EMP_PROJ relation in Figures 14.10(a) and (b), whereas Pnumber is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber. To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2, as shown in Figure 14.10(c).

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an

(a)

EMP_PROJ

Ssn	Ename	Projs	
		Pnumber	Hours

(b)

EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

Figure 14.10
Normalizing nested
relations into 1NF.
(a) Schema of the
EMP_PROJ relation with
a *nested relation* attribute
PROJS. (b) Sample
extension of the
EMP_PROJ relation
showing nested relations
within each tuple.
(c) Decomposition of
EMP_PROJ into relations
EMP_PROJ1 and
EMP_PROJ2 by
propagating the primary
key.

(c)

EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

unnormalized relation schema with many levels of nesting into 1NF relations. As an example, consider the following:

CANDIDATE (Ssn, Name, {JOB_HIST (Company, Highest_position,
    {SAL_HIST (Year, Max_sal)})})

The foregoing describes data about candidates applying for jobs with their job history as a nested relation within which the salary history is stored as a deeper nested

relation. The first normalization using internal partial keys Company and Year, respectively, results in the following 1NF relations:

> CANDIDATE_1 (Ssn, Name)
> CANDIDATE_JOB_HIST (<u>Ssn, Company</u>, Highest_position)
> CANDIDATE_SAL_HIST (<u>Ssn, Company, Year</u>, Max-sal)

The existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

> PERSON (<u>Ss#</u>, {Car_lic#}, {Phone#})

This relation represents the fact that a person has multiple cars and multiple phones. If strategy 2 above is followed, it results in an all-key relation:

> PERSON_IN_1NF (<u>Ss#, Car_lic#, Phone#</u>)

To avoid introducing any extraneous relationship between Car_lic# and Phone#, all possible combinations of values are represented for every Ss#, giving rise to redundancy. This leads to the problems that are typically discovered at a later stage of normalization and that are handled by multivalued dependencies and 4NF, which we will discuss in Section 14.6. The right way to deal with the two multivalued attributes in PERSON shown previously is to decompose it into two separate relations, using strategy 1 discussed above: P1(<u>Ss#, Car_lic#</u>) and P2(<u>Ss#, Phone#</u>).

A note about the relations that involve attributes that go beyond just numeric and character string data. It is becoming common in today's databases to incorporate images, documents, video clips, audio clips, and so on. When these are stored in a relation, the entire object or file is treated as an atomic value, which is stored as a BLOB (binary large object) or CLOB (character large object) data type using SQL. For practical purposes, the object is treated as an atomic, single-valued attribute and hence it maintains the 1NF status of the relation.

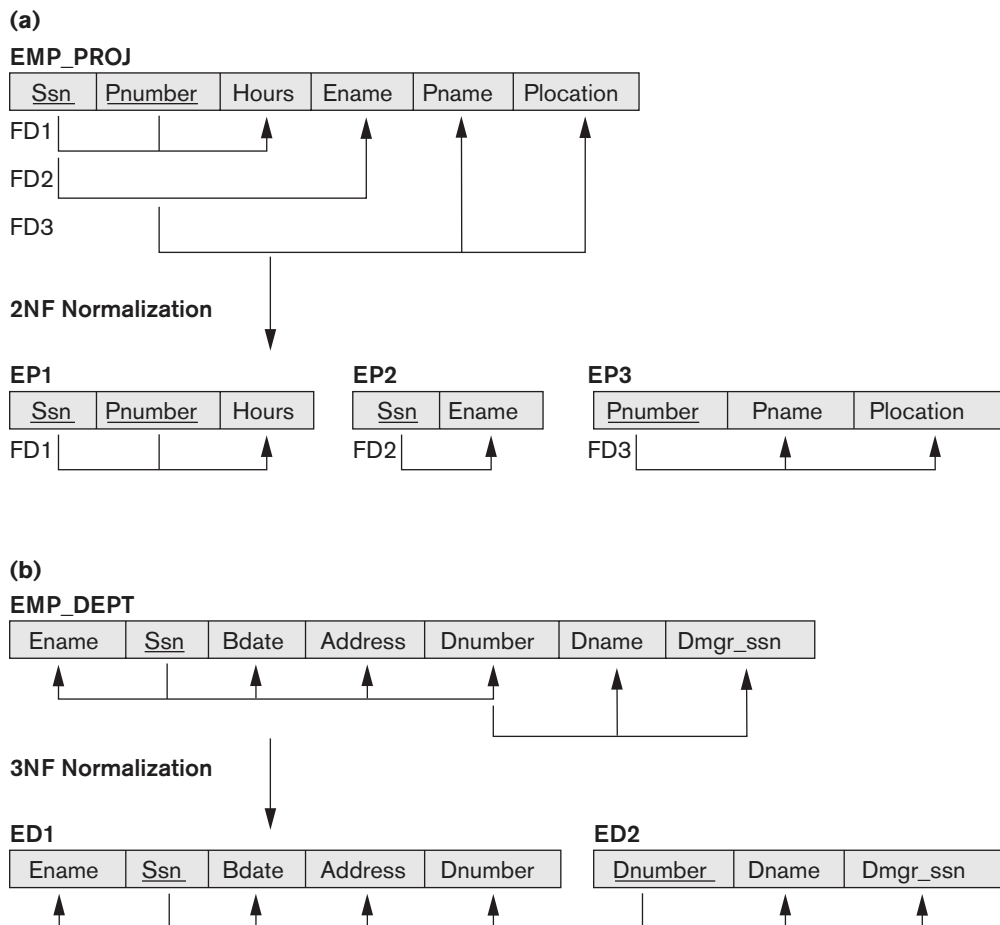### 14.3.5 Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold anymore; that is, for any attribute $A \, \varepsilon \, X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \, \varepsilon \, X$ can be removed from X and the dependency still holds; that is, for some $A \, \varepsilon \, X$, $(X - \{A\}) \rightarrow Y$. In Figure 14.3(b), {Ssn, Pnumber} $\rightarrow$ Hours is a full dependency (neither Ssn $\rightarrow$ Hours nor Pnumber $\rightarrow$ Hours holds). However, the dependency {Ssn, Pnumber} $\rightarrow$ Ename is partial because Ssn $\rightarrow$ Ename holds.

> **Definition.** A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 14.3(b) is in

1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. Each of the functional dependencies FD2 and FD3 violates 2NF because Ename can be functionally determined by only Ssn, and both Pname and Plocation can be functionally determined by only Pnumber. Attributes Ssn and Pnumber are a part of the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure 14.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 14.11(a), each of which is in 2NF.

Figure 14.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

14.3.6 Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R,[11] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency Ssn $\rightarrow$ Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 14.3(a), because both the dependencies Ssn $\rightarrow$ Dnumber and Dnumber $\rightarrow$ Dmgr_ssn hold *and* Dnumber is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of Dmgr_ssn on Dnumber is undesirable in EMP_DEPT since Dnumber is not a key of EMP_DEPT.

> **Definition.** According to Codd's original definition, a relation schema R is in **3NF** if it satisfies 2NF *and* no nonprime attribute of R is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 14.3(a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 14.11(b). Intuitively, we see that ED1 and ED2 represent independent facts about employees and departments, both of which are entities in their own right. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a *problematic* FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Moreover, the general definition of 3NF we present in Section 14.4.2 automatically covers the condition that the relation also satisfies 2NF. Table 14.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding *remedy* or normalization performed to achieve the normal form.

14.4 General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies discussed in Section 14.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on

---

[11]This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where X is the primary key but Z may be (a subset of) a candidate key.

Table 14.1   Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

the *primary key.* The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined. These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.
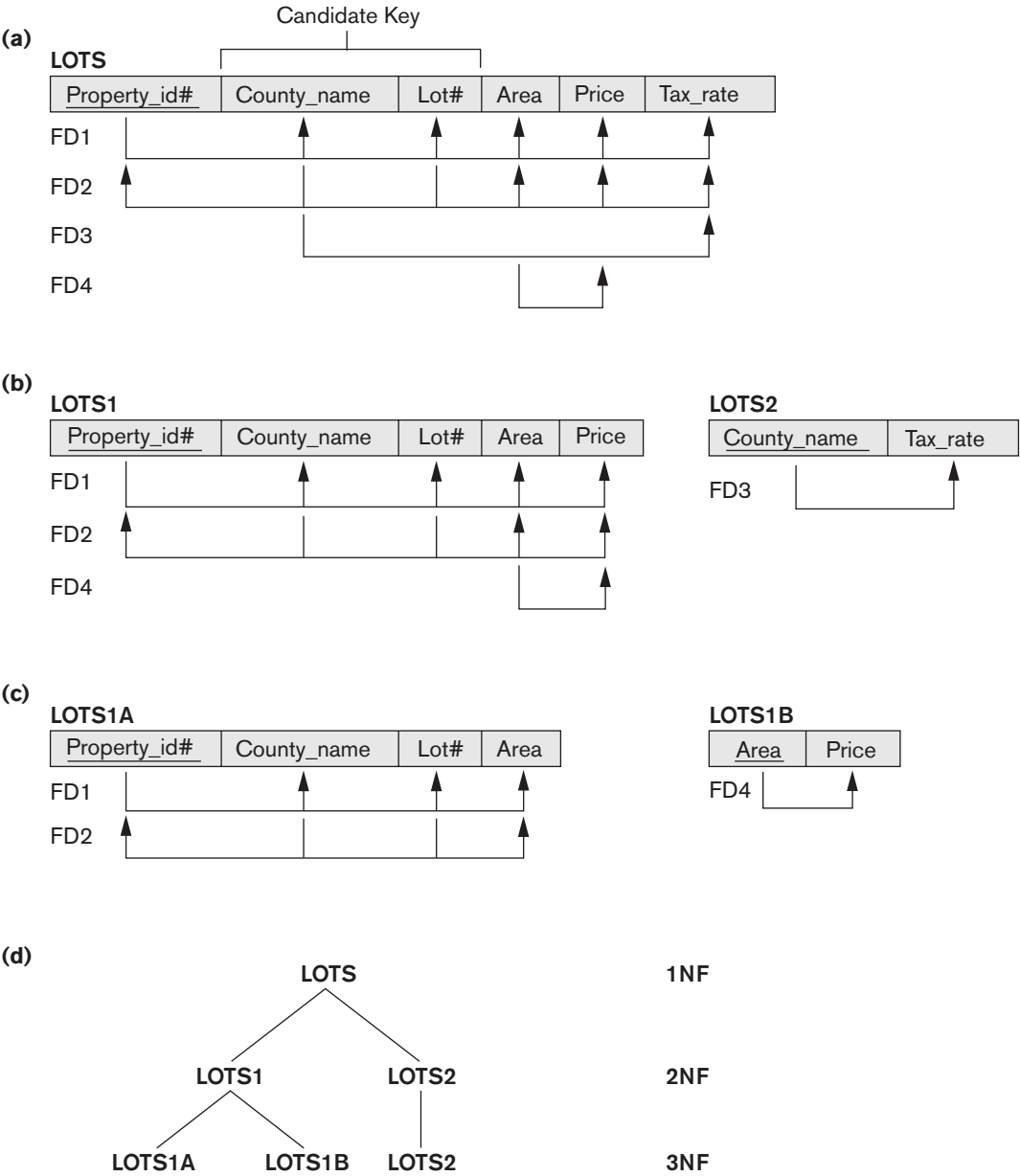
14.4.1 General Definition of Second Normal Form

> **Definition.** A relation schema R is in **second normal form (2NF)** if every nonprime attribute A in R is not partially dependent on *any* key of R.[12]

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 14.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

---

[12]This definition can be restated as follows: A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on *every* key of R.

Figure 14.12
Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies
FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2.
(c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Progressive
normalization of LOTS into a 3NF design.

Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 in Figure 14.12(a) hold. We choose Property_id# as the primary key, so it is underlined in Figure 14.12(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate
FD4: Area → Price

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), whereas FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 14.12(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

14.4.2 General Definition of Third Normal Form

Definition. A relation schema R is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.[13]

According to this definition, LOTS2 (Figure 14.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 14.12(c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because Price is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute Area.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. In other words, if a relation passes the general 3NF test, then it automatically passes the 2NF test.

---

[13]Note that based on inferred f.d.'s (which are discussed in Section 15.1), the f.d. $Y \rightarrow YA$ also holds whenever $Y \rightarrow A$ is true. Therefore, a slightly better way of saying this statement is that {A-X} is a prime attribute of R.

If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF by the general definition above because the LHS County_name in FD3 is not a superkey. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed *in any order*.

14.4.3 Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in R that meets either of the two conditions, namely (a) and (b). The first condition "catches" two types of problematic dependencies:

- A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.
- A proper subset of a key of R functionally determines a nonprime attribute. Here we have a partial dependency that violates 2NF.

Thus, condition (a) alone addresses the problematic dependencies that were causes for second and third normalization as we discussed.

Therefore, we can state a **general alternative definition of 3NF** as follows:

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:
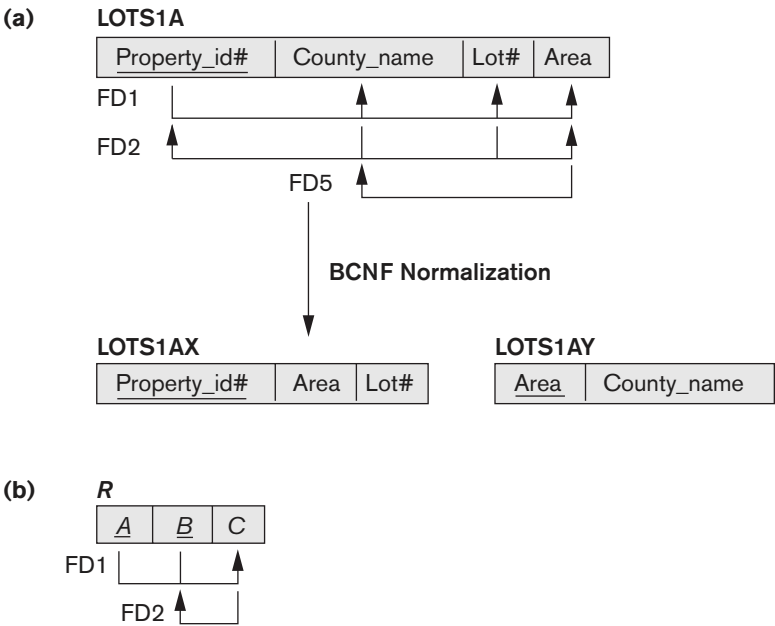
- It is fully functionally dependent on every key of R.
- It is nontransitively dependent on every key of R.

However, note the clause (b) in the general definition of 3NF. It allows certain functional dependencies to slip through or escape in that they are OK with the 3NF definition and hence are not "caught" by the 3NF definition even though they may be potentially problematic. The Boyce-Codd normal form "catches" these dependencies in that it does not allow them. We discuss that normal form next.

14.5 Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. We pointed out in the last subsection that although 3NF allows functional dependencies that conform to the clause (b) in the 3NF definition, BCNF disallows them and hence is a stricter definition of a normal form.

Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in Figure 14.12(a) with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County

Figure 14.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d. $C \rightarrow B$.

are restricted to 1.1, 1.2, ... , 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: Area $\rightarrow$ County_name. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because this f.d. conforms to clause (b) in the general definition of 3NF, County_name being a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation R(Area, County_name), since there are only 16 possible Area values (see Figure 14.13). This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

> **Definition.** A relation schema R is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, then X is a superkey of R.

The formal definition of BCNF differs from the definition of 3NF in that clause (b) of 3NF, which allows f.d.'s having the RHS as a prime attribute, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because Area is not a superkey of LOTS1A. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 14.13(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if there exists some f.d. $X \rightarrow A$ that holds in a relation schema R with X not being a superkey

and A being a prime attribute will *R* be in 3NF but not in BCNF. The relation schema *R* shown in Figure 14.13(b) illustrates the general case of such a relation. Such an f.d. leads to potential redundancy of data, as we illustrated above in case of FD5: Area → County_name.in LOTS1A relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization status of just 1NF or 2NF is not considered adequate, since both were developed historically to be intermediate normal forms as stepping stones to 3NF and BCNF.

14.5.1 Decomposition of Relations not in BCNF

As another example, consider Figure 14.14, which shows a relation TEACH with the following dependencies:

FD1:   {Student, Course} → Instructor
FD2:[14]   Instructor → Course

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 14.13(b), with Student as *A*, Course as *B*, and Instructor as *C*. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. R1 (Student, Instructor) and R2(Student, Course)
2. R1 (Course, Instructor) and R2(Course, Student)
3. R1 (Instructor, Course) and R2(Instructor, Student)

All three decompositions *lose* the functional dependency FD1. The question then becomes: Which of the above three is a *desirable decomposition*? As we pointed out earlier (Section 14.3.1), we strive to meet two properties of decomposition during

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Figure 14.14
A relation TEACH that is in 3NF but not BCNF.

---

[14]This dependency means that *each instructor teaches one course* is a constraint for this application.

the normalization process: the nonadditive join property and the functional dependency preservation property. We are not able to meet the functional dependency preservation for any of the above BCNF decompositions as seen above; but we must meet the nonadditive join property. A simple test comes in handy to test the binary decomposition of a relation into two relations:

NJB (Nonadditive Join Test for Binary Decompositions). A decomposition $D = \{R_1, R_2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R *if and only if* either

- The FD $((R_1 \cap R_2) \to (R_1 - R_2))$ is in F^{+}[15], or
- The FD $((R_1 \cap R_2) \to (R_2 - R_1))$ is in F^{+}

If we apply this test to the above three decompositions, we find that only the third decomposition meets the test. In the third decomposition, the $R_1 \cap R_2$ for the above test is Instructor and $R_1 - R_2$ is Course. Because Instructor $\to$ Course, the NJB test is satisfied and the decomposition is nonadditive. (It is left as an exercise for the reader to show that the first two decompositions do not meet the NJB test.) Hence, the proper decomposition of TEACH into BCNF relations is:

TEACH1 (<u>Instructor</u>, Course) and TEACH2 (<u>Instructor</u>, <u>Student</u>)

We make sure that we meet this property, because nonadditive decomposition is a must during normalization. You should verify that this property holds with respect to our informal successive normalization examples in Sections 14.3 and 14.4 and also by the decomposition of LOTS1A into two BCNF relations LOTS1AX and LOTS1AY.

In general, a relation R not in BCNF can be decomposed so as to meet the nonadditive join property by the following procedure.[16] It decomposes R successively into a set of relations that are in BCNF:

Let R be the relation not in BCNF, let $X \subseteq R$, and let $X \to A$ be the FD that causes a violation of BCNF. R may be decomposed into two relations:

$R - A$

XA

If either $R - A$ or XA. is not in BCNF, repeat the process.

The reader should verify that if we applied the above procedure to LOTS1A, we obtain relations LOTS1AX and LOTS1AY as before. Similarly, applying this procedure to TEACH results in relations TEACH1 and TEACH2

---

[15]The notation F^{+} refers to the cover of the set of functional dependencies and includes all f.d.'s implied by F. It is discussed in detail in Section 15.1. Here, it is enough to make sure that one of the two f.d.'s actually holds for the nonadditive decomposition into R_1 and R_2 to pass this test.

[16]Note that this procedure is based on Algorithm 15.5 from Chapter 15 for producing BCNF schemas by decomposition of a universal schema.

Note that if we designate (Student, Instructor) as a primary key of the relation TEACH, the FD instructor → Course causes a partial (non-fully-functional) dependency of Course on a part of this key. This FD may be removed as a part of second normalization (or by a direct application of the above procedure to achieve BCNF) yielding exactly the same two relations in the result. This is an example of a case where we may reach the same ultimate BCNF design via alternate paths of normalization.

14.6 Multivalued Dependency and Fourth Normal Form

Consider the relation EMP shown in Figure 14.15(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another.[17] To keep the relation state consistent and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. In the relation state shown in Figure 14.15(a), the employee with Ename Smith works on two projects 'X' and 'Y' and has two dependents 'John' and 'Anna', and therefore there are four tuples to represent these facts together. The relation EMP is an **all-key relation** (with key made up of all attributes) and therefore has no f.d.'s and as such qualifies to be a BCNF relation. We can see that there is an obvious redundancy in the relation EMP—the dependent information is repeated for every project and the project information is repeated for every dependent.

As illustrated by the EMP relation, some relations have constraints that cannot be specified as functional dependencies and hence are not in violation of BCNF. To address this situation, the concept of *multivalued dependency* (MVD) was proposed and, based on this dependency, the *fourth normal form* was defined. A more formal discussion of MVDs and their properties is deferred to Chapter 15. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 14.3.4), which disallows an attribute in a tuple to have a *set of values*. If more than one multivalued attribute is present, the second option of normalizing the relation (see Section 14.3.4) introduces a multivalued dependency. Informally, whenever two *independent* 1:N relationships A:B and A:C are mixed in the same relation, R(A, B, C), an MVD may arise.[18]

14.6.1 Formal Definition of Multivalued Dependency

> **Definition.** A multivalued dependency $X \longrightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any

---

[17]In an ER diagram, each would be represented as a multivalued attribute or as a weak entity type (see Chapter 7).

[18]This MVD is denoted as $A \longrightarrow B|C$.

(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(c) SUPPLY

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

(d) R_1

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Figure 14.15
Fourth and fifth normal forms.
(a) The EMP relation with two MVDs: Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname.
(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and
EMP_DEPENDENTS.
(c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R_1, R_2, R_3).
(d) Decomposing the relation SUPPLY into the 5NF relations R_1, R_2, R_3.

relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties,[19] where we use Z to denote $(R - (X \cup Y))$:[20]

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$

---

[19]The tuples t_1, t_2, t_3, and t_4 are not necessarily distinct.

[20]Z is shorthand for the attributes in R after the attributes in $(X \cup Y)$ are removed from R.

Whenever $X \twoheadrightarrow Y$ holds, we say that X **multidetermines** Y. Because of the symmetry in the definition, whenever $X \twoheadrightarrow Y$ holds in R, so does $X \twoheadrightarrow Z$. Hence, $X \twoheadrightarrow Y$ implies $X \twoheadrightarrow Z$ and therefore it is sometimes written as $X \twoheadrightarrow Y|Z$.

An MVD $X \twoheadrightarrow Y$ in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) $X \cup Y = R$. For example, the relation EMP_PROJECTS in Figure 14.15(b) has the trivial MVD Ename $\twoheadrightarrow$ Pname and the relation EMP_DEPENDENTS has the trivial MVD Ename $\twoheadrightarrow$ Dname. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. A trivial MVD will hold in *any* relation state r of R; it is called trivial because it does not specify any significant or meaningful constraint on R.

If we have a *nontrivial MVD* in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure 14.15(a), the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname). This redundancy is clearly undesirable. However, the EMP schema is in BCNF because *no* functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP. Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. Furthermore, it is rare that such all-key relations with a combinatorial occurrence of repeated values would be designed in practice. However, recognition of MVDs as a potential problematic dependency is essential in relational design.

We now present the definition of **fourth normal form (4NF)**, which is violated when a relation has undesirable multivalued dependencies and hence can be used to identify and decompose such relations.

> **Definition.** A relation schema R is in 4**NF** with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in F^{+},[21] X is a superkey for R.

We can state the following points:

- An all-key relation is always in BCNF since it has no FDs.
- An all-key relation such as the EMP relation in Figure 14.15(a), which has no FDs but has the MVD Ename $\twoheadrightarrow$ Pname | Dname, is not in 4NF.
- A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
- The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure 14.15(a). EMP is not in 4NF because in the nontrivial MVDs Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname,

---

[21]F^{+} refers to the cover of functional dependencies F, or all dependencies that are implied by F. This is defined in Section 15.1.

and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 14.15(b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename $\twoheadrightarrow$ Pname in EMP_PROJECTS and Ename $\twoheadrightarrow$ Dname in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

14.7 Join Dependencies and Fifth Normal Form

In our discussion so far, we have pointed out the problematic functional dependencies and shown how they were eliminated by a process of repeated binary decomposition during the process of normalization to achieve 1NF, 2NF, 3NF, and BCNF. These binary decompositions must obey the NJB property for which we introduced a test in Section 14.5 while discussing the decomposition to achieve BCNF. Achieving 4NF typically involves eliminating MVDs by repeated binary decompositions as well. However, in some cases there may be no nonadditive join decomposition of *R* into *two* relation schemas, but there may be a nonadditive join decomposition into *more than two* relation schemas. Moreover, there may be no functional dependency in *R* that violates any normal form up to BCNF, and there may be no nontrivial MVD present in *R* either that violates 4NF. We then resort to another dependency called the *join dependency* and, if it is present, carry out a *multiway decomposition* into fifth normal form (5NF). It is important to note that such a dependency is a peculiar semantic constraint that is difficult to detect in practice; therefore, normalization into 5NF is rarely done in practice.

> **Definition.** A **join dependency** (**JD**), denoted by JD(R_1, R_2, … , R_n), specified on relation schema *R*, specifies a constraint on the states *r* of *R*. The constraint states that every legal state *r* of *R* should have a nonadditive join decomposition into R_1, R_2, … , R_n. Hence, for every such *r* we have
>
> $$* (\pi_{R_1}(r), \pi_{R_2}(r), … , \pi_{R_n}(r)) = r$$

Notice that an MVD is a special case of a JD where *n* = 2. That is, a JD denoted as JD(R_1, R_2) implies an MVD ($R_1 \cap R_2$) $\twoheadrightarrow$ ($R_1 - R_2$)(or, by symmetry, ($R_1 \cap R_2$) $\twoheadrightarrow$ ($R_2 - R_1$)). A join dependency JD(R_1, R_2, … , R_n), specified on relation schema *R*, is a **trivial** JD if one of the relation schemas R_i in JD(R_1, R_2, … , R_n) is equal to *R*. Such a dependency is called trivial because it has the nonadditive join property for any relation state *r* of *R* and thus does not specify any constraint on *R*. We can now define the fifth normal form, which is also called *project-join normal form*.

> **Definition.** A relation schema *R* is in **fifth normal form (5NF)** (or **project-join normal form (PJNF)**) with respect to a set *F* of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, … , R_n) in F^+ (that is, implied by *F*),[22] every R_i is a superkey of *R*.

---

[22]Again, F^+ refers to the cover of functional dependencies *F*, or all dependencies that are implied by *F*. This is defined in Section 15.1.

For an example of a JD, consider once again the SUPPLY all-key relation in Figure 14.15(c). Suppose that the following additional constraint always holds: Whenever a supplier *s* supplies part *p*, *and* a project *j* uses part *p*, *and* the supplier *s* supplies *at least one* part to project *j*, *then* supplier *s* will also be supplying part *p* to project *j*. This constraint can be restated in other ways and specifies a join dependency $JD(R_1, R_2, R_3)$ among the three projections R_1 (Sname, Part_name), R_2 (Sname, Proj_name), and R_3 (Part_name, Proj_name) of SUPPLY. If this constraint holds, the tuples below the dashed line in Figure 14.15(c) must exist in any legal state of the SUPPLY relation that also contains the tuples above the dashed line. Figure 14.15(d) shows how the SUPPLY relation *with the join dependency* is decomposed into three relations R_1, R_2, and R_3 that are each in 5NF. Notice that applying a natural join to *any two* of these relations *produces spurious tuples,* but applying a natural join to *all three together* does not. The reader should verify this on the sample relation in Figure 14.15(c) and its projections in Figure 14.15(d). This is because only the JD exists, but no MVDs are specified. Notice, too, that the $JD(R_1, R_2, R_3)$ is specified on *all* legal relation states, not just on the one shown in Figure 14.15(c).

Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer. Therefore, the current practice of database design pays scant attention to them. One result due to Date and Fagin (1992) relates to conditions detected using f.d.'s alone and ignores JDs completely. It states: "If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF."

14.8 Summary

In this chapter we discussed several pitfalls in relational database design using intuitive arguments. We identified informally some of the measures for indicating whether a relation schema is *good* or *bad*, and we provided informal guidelines for a good design. These guidelines are based on doing a careful conceptual design in the ER and EER model, following the mapping procedure in Chapter 9 to map entities and relationships into relations. Proper enforcement of these guidelines and lack of redundancy will avoid the insertion/deletion/update anomalies and generation of spurious data. We recommended limiting NULL values, which cause problems during SELECT, JOIN, and aggregation operations. Then we presented some formal concepts that allow us to do relational design in a top-down fashion by analyzing relations individually. We defined this process of design by analysis and decomposition by introducing the process of normalization.

We defined the concept of functional dependency, which is the basic tool for analyzing relational schemas, and we discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. Next we described the normalization process for achieving good designs by testing relations for undesirable types of *problematic* functional dependencies. We provided a treatment of successive normalization based on a predefined primary key in each relation, and we then relaxed this requirement and provided more

general definitions of second normal form (2NF) and third normal form (3NF) that take all candidate keys of a relation into account. We presented examples to illustrate how, by using the general definition of 3NF, a given relation may be analyzed and decomposed to eventually yield a set of relations in 3NF.

We presented Boyce-Codd normal form (BCNF) and discussed how it is a stronger form of 3NF. We also illustrated how the decomposition of a non-BCNF relation must be done by considering the nonadditive decomposition requirement. We presented a test for the nonadditive join property of binary decompositions and also gave a general algorithm to convert any relation not in BCNF into a set of BCNF relations. We motivated the need for an additional constraint beyond the functional dependencies based on mixing of independent multivalued attributes into a single relation. We introduced multivalued dependency (MVD) to address such conditions and defined the fourth normal form based on MVDs. Finally, we introduced the fifth normal form, which is based on join dependency and which identifies a peculiar constraint that causes a relation to be decomposed into several components so that they always yield the original relation after a join. In practice, most commercial designs have followed the normal forms up to BCNF. The need to decompose into 5NF rarely arises in practice, and join dependencies are difficult to detect for most practical situations, making 5NF more of theoretical value.

Chapter 15 presents synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we discuss the concepts of *nonadditive* (or *lossless*) *join* and *dependency preservation,* which are enforced by some of these algorithms. Other topics in Chapter 15 include a more detailed treatment of functional and multivalued dependencies, and other types of dependencies.

Review Questions

14.1. Discuss attribute semantics as an informal measure of goodness for a relation schema.

14.2. Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.

14.3. Why should NULLs in a relation be avoided as much as possible? Discuss the problem of spurious tuples and how we may prevent it.

14.4. State the informal guidelines for relation schema design that we discussed. Illustrate how violation of these guidelines may be harmful.

14.5. What is a functional dependency? What are the possible sources of the information that defines the functional dependencies that hold among the attributes of a relation schema?

14.6. Why can we not infer a functional dependency automatically from a particular relation state?

14.7. What does the term *unnormalized relation* refer to? How did the normal forms develop historically from first normal form up to Boyce-Codd normal form?

14.8. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?

14.9. What undesirable dependencies are avoided when a relation is in 2NF?

14.10. What undesirable dependencies are avoided when a relation is in 3NF?

14.11. In what way do the generalized definitions of 2NF and 3NF extend the definitions beyond primary keys?

14.12. Define *Boyce-Codd normal form*. How does it differ from 3NF? Why is it considered a stronger form of 3NF?

14.13. What is multivalued dependency? When does it arise?

14.14. Does a relation with two or more columns always have an MVD? Show with an example.

14.15. Define *fourth normal form*. When is it violated? When is it typically applicable?

14.16. Define *join dependency* and *fifth normal form*.

14.17. Why is 5NF also called project-join normal form (PJNF)?

14.18. Why do practical database designs typically aim for BCNF and not aim for higher normal forms?

Exercises

14.19. Suppose that we have the following requirements for a university database that is used to keep track of students' transcripts:

a. The university keeps track of each student's name (Sname), student number (Snum), Social Security number (Ssn), current address (Sc_addr) and phone (Sc_phone), permanent address (Sp_addr) and phone (Sp_phone), birth date (Bdate), sex (Sex), class (Class) ('freshman', 'sophomore', … , 'graduate'), major department (Major_code), minor department (Minor_code) (if any), and degree program (Prog) ('b.a.', 'b.s.', … , 'ph.d.'). Both Ssn and student number have unique values for each student.

b. Each department is described by a name (Dname), department code (Dcode), office number (Doffice), office phone (Dphone), and college (Dcollege). Both name and code have unique values for each department.

c. Each course has a course name (Cname), description (Cdesc), course number (Cnum), number of semester hours (Credit), level (Level), and offering department (Cdept). The course number is unique for each course.

d. Each section has an instructor (Iname), semester (Semester), year (Year), course (Sec_course), and section number (Sec_num). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, … , up to the total number of sections taught during each semester.

e. A grade record refers to a student (Ssn), a particular section, and a grade (Grade).

Design a relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to render the specification complete.

14.20. What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 14.3 and 14.4?

14.21. In what normal form is the LOTS relation schema in Figure 14.12(a) with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

14.22. Prove that any relation schema with two attributes is in BCNF.

14.23. Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_ LOCS relations in Figure 14.5 (result shown in Figure 14.6)?

14.24. Consider the universal relation R = {$A, B, C, D, E, F, G, H, I, J$} and the set of functional dependencies F = {{A, B}→{C}, {A}→{D, E}, {B}→{F}, {F}→{G, H}, {D}→{I, J}}. What is the key for R? Decompose R into 2NF and then 3NF relations.

14.25. Repeat Exercise 14.24 for the following different set of functional dependencies G = {{A, B}→{C}, {B, D}→{E, F}, {A, D}→{G, H}, {A}→{I}, {H}→{J}}.

14.26. Consider the following relation:

A	B	C	TUPLE#
10	b1	c1	1
10	b2	c2	2
11	b4	c1	3
12	b3	c4	4
13	b1	c1	5
14	b3	c4	6

a. Given the previous extension (state), which of the following dependencies *may hold* in the above relation? If the dependency cannot hold, explain why *by specifying the tuples that cause the violation.*

i. $A \rightarrow B,$ ii. $B \rightarrow C,$ iii. $C \rightarrow B,$ iv. $B \rightarrow A,$ v. $C \rightarrow A$

b. Does the above relation have a potential candidate key? If it does, what is it? If it does not, why not?

14.27. Consider a relation $R(A, B, C, D, E)$ with the following dependencies:

$$AB \rightarrow C,\ CD \rightarrow E,\ DE \rightarrow B$$

Is AB a candidate key of this relation? If not, is ABD? Explain your answer.

14.28. Consider the relation R, which has attributes that hold schedules of courses and sections at a university; R = {Course_no, Sec_no, Offering_dept, Credit_hours, Course_level, Instructor_ssn, Semester, Year, Days_hours, Room_no, No_of_students}. Suppose that the following functional dependencies hold on R:

{Course_no} → {Offering_dept, Credit_hours, Course_level}
{Course_no, Sec_no, Semester, Year} → {Days_hours, Room_no,
    No_of_students, Instructor_ssn}
{Room_no, Days_hours, Semester, Year} → {Instructor_ssn, Course_no,
    Sec_no}

Try to determine which sets of attributes form keys of R. How would you normalize this relation?

14.29. Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (O#, Odate, Cust#, Total_amount)
ORDER_ITEM(O#, I#, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The Total_price refers to one item, Odate is the date on which the order was placed, and the Total_amount is the amount of the order. If we apply a natural join on the relations ORDER_ITEM and ORDER in this database, what does the resulting relation schema RES look like? What will be its key? Show the FDs in this resulting relation. Is RES in 2NF? Is it in 3NF? Why or why not? (State assumptions, if you make any.)

14.30. Consider the following relation:

   CAR_SALE(Car#, Date_sold, Salesperson#, Commission%, Discount_amt)

Assume that a car may be sold by multiple salespeople, and hence {Car#, Salesperson#} is the primary key. Additional dependencies are

Date_sold → Discount_amt and
Salesperson# → Commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

14.31. Consider the following relation for published books:

BOOK (Book_title, Author_name, Book_type, List_price, Author_affil,
    Publisher)

Author_affil refers to the affiliation of author. Suppose the following dependencies exist:

Book_title → Publisher, Book_type
Book_type → List_price
Author_name → Author_affil

a. What normal form is the relation in? Explain your answer.

b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

14.32. This exercise asks you to convert business statements into dependencies. Consider the relation DISK_DRIVE (Serial_number, Manufacturer, Model, Batch, Capacity, Retailer). Each tuple in the relation DISK_DRIVE contains information about a disk drive with a unique Serial_number, made by a manufacturer, with a particular model number, released in a certain batch, which has a certain storage capacity and is sold by a certain retailer. For example, the tuple Disk_drive ('1978619', 'WesternDigital', 'A2235X', '765234', 500, 'CompUSA') specifies that WesternDigital made a disk drive with serial number 1978619 and model number A2235X, released in batch 765234; it is 500GB and sold by CompUSA.

Write each of the following dependencies as an FD:

a. The manufacturer and serial number uniquely identifies the drive.

b. A model number is registered by a manufacturer and therefore can't be used by another manufacturer.

c. All disk drives in a particular batch are the same model.

d. All disk drives of a certain model of a particular manufacturer have exactly the same capacity.

14.33. Consider the following relation:

R (Doctor#, Patient#, Date, Diagnosis, Treat_code, Charge)

In the above relation, a tuple describes a visit of a patient to a doctor along with a treatment code and daily charge. Assume that diagnosis is determined (uniquely) for each patient by a doctor. Assume that each treatment code has a fixed charge (regardless of patient). Is this relation in 2NF? Justify your answer and decompose if necessary. Then argue whether further normalization to 3NF is necessary, and if so, perform it.

14.34. Consider the following relation:

CAR_SALE (Car_id, Option_type, Option_listprice, Sale_date, Option_discountedprice)

This relation refers to options installed in cars (e.g., cruise control) that were sold at a dealership, and the list and discounted prices of the options.

If CarID → Sale_date and Option_type → Option_listprice and CarID, Option_type → Option_discountedprice, argue using the generalized definition of the 3NF

that this relation is not in 3NF. Then argue from your knowledge of 2NF, why it is not even in 2NF.

14.35. Consider the relation:

BOOK (Book_Name, Author, Edition, Year)

with the data:

Book_Name	Author	Edition	Copyright_Year
DB_fundamentals	Navathe	4	2004
DB_fundamentals	Elmasri	4	2004
DB_fundamentals	Elmasri	5	2007
DB_fundamentals	Navathe	5	2007

a. Based on a common-sense understanding of the above data, what are the possible candidate keys of this relation?

b. Justify that this relation has the MVD {Book} $\twoheadrightarrow$ {Author} | {Edition, Year}.

c. What would be the decomposition of this relation based on the above MVD? Evaluate each resulting relation for the highest normal form it possesses.

14.36. Consider the following relation:

TRIP (Trip_id, Start_date, Cities_visited, Cards_used)

This relation refers to business trips made by company salespeople. Suppose the TRIP has a single Start_date but involves many Cities and salespeople may use multiple credit cards on the trip. Make up a mock-up population of the table.

a. Discuss what FDs and/or MVDs exist in this relation.

b. Show how you will go about normalizing the relation.

Laboratory Exercises

Note: The following exercises use the DBD (Data Base Designer) system that is described in the laboratory manual.

The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for this problem are coded as:

$$R = [a, b, c, d, e, f, g, h, i, j]$$
$$F = [[[a, b],[c]],$$
$$[[a],[d, e]],$$
$$[[b],[f]],$$
$$[[f],[g, h]],$$
$$[[d],[i, j]]]$$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

14.37. Using the DBD system, verify your answers to the following exercises:

    a. 14.24 (3NF only)

    b. 14.25

    c. 14.27

    d. 14.28

Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies. Date and Fagin (1992) give some simple and practical results related to higher normal forms.

Additional references to relational design theory are given in Chapter 15.

Relational Database Design Algorithms and Further Dependencies

Chapter 14 presented a **top-down relational design** technique and related concepts used extensively in commercial database design projects today. The procedure involves designing an ER or EER conceptual schema and then mapping it to the relational model by a procedure such as the one described in Chapter 9. Primary keys are assigned to each relation based on known functional dependencies. In the subsequent process, which may be called **relational design by analysis**, initially designed relations from the above procedure—or those inherited from previous files, forms, and other sources—are analyzed to detect undesirable functional dependencies. These dependencies are removed by the successive normalization procedure that we described in Section 14.3 along with definitions of related normal forms, which are successively better states of design of individual relations. In Section 14.3 we assumed that primary keys were assigned to individual relations; in Section 14.4 a more general treatment of normalization was presented where all candidate keys are considered for each relation, and Section 14.5 discussed a further normal form called BCNF. Then in Sections 14.6 and 14.7 we discussed two more types of dependencies—multivalued dependencies and join dependencies—that can also cause redundancies and showed how they can be eliminated with further normalization.

In this chapter, we use the theory of normal forms and functional, multivalued, and join dependencies developed in the last chapter and build upon it while maintaining three different thrusts. First, we discuss the concept of inferring new functional dependencies from a given set and discuss notions including closure, cover, minimal cover, and equivalence. Conceptually, we need to capture the semantics of

attributes within a relation completely and succinctly, and the minimal cover allows us to do it. Second, we discuss the desirable properties of nonadditive (lossless) joins and preservation of functional dependencies. A general algorithm to test for nonadditivity of joins among a set of relations is presented. Third, we present an approach to **relational design by synthesis** of functional dependencies. This is a **bottom-up approach to design** that presupposes that the known functional dependencies among sets of attributes in the Universe of Discourse (UoD) have been given as input. We present algorithms to achieve the desirable normal forms, namely 3NF and BCNF, and achieve one or both of the desirable properties of nonadditivity of joins and functional dependency preservation. Although the synthesis approach is theoretically appealing as a formal approach, it has not been used in practice for large database design projects because of the difficulty of providing all possible functional dependencies up front before the design can be attempted. Alternately, with the approach presented in Chapter 14, successive decompositions and ongoing refinements to design become more manageable and may evolve over time. The final goal of this chapter is to discuss further the multivalued dependency (MVD) concept we introduced in Chapter 14 and briefly point out other types of dependencies that have been identified.

In Section 15.1 we discuss the rules of inference for functional dependencies and use them to define the concepts of a cover, equivalence, and minimal cover among functional dependencies. In Section 15.2, first we describe the two desirable **properties of decompositions**, namely, the dependency preservation property and the nonadditive (or lossless) join property, which are both used by the design algorithms to achieve desirable decompositions. It is important to note that it is *insufficient* to test the relation schemas *independently of one another* for compliance with higher normal forms like 2NF, 3NF, and BCNF. The resulting relations must collectively satisfy these two additional properties to qualify as a good design. Section 15.3 is devoted to the development of relational design algorithms that start off with one giant relation schema called the **universal relation**, which is a hypothetical relation containing all the attributes. This relation is decomposed (or in other words, the given functional dependencies are synthesized) into relations that satisfy a certain normal form like 3NF or BCNF and also meet one or both of the desirable properties.

In Section 15.5 we discuss the multivalued dependency (MVD) concept further by applying the notions of inference, and equivalence to MVDs. Finally, in Section 15.6 we complete the discussion on dependencies among data by introducing inclusion dependencies and template dependencies. Inclusion dependencies can represent referential integrity constraints and class/subclass constraints across relations. We also describe some situations where a procedure or function is needed to state and verify a functional dependency among attributes. Then we briefly discuss domain-key normal form (DKNF), which is considered the most general normal form. Section 15.7 summarizes this chapter.

It is possible to skip some or all of Sections 15.3, 15.4, and 15.5 in an introductory database course.

15.1 Further Topics in Functional Dependencies: Inference Rules, Equivalence, and Minimal Cover

We introduced the concept of functional dependencies (FDs) in Section 14.2, illustrated it with some examples, and developed a notation to denote multiple FDs over a single relation. We identified and discussed problematic functional dependencies in Sections 14.3 and 14.4 and showed how they can be eliminated by a proper decomposition of a relation. This process was described as *normalization*, and we showed how to achieve the first through third normal forms (1NF through 3NF) given primary keys in Section 14.3. In Sections 14.4 and 14.5 we provided generalized tests for 2NF, 3NF, and BCNF given any number of candidate keys in a relation and showed how to achieve them. Now we return to the study of functional dependencies and show how new dependencies can be inferred from a given set and discuss the concepts of closure, equivalence, and minimal cover that we will need when we later consider a synthesis approach to design of relations given a set of FDs.

15.1.1 Inference Rules for Functional Dependencies

We denote by F the set of functional dependencies that are specified on relation schema R. Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F. Those other dependencies can be *inferred* or *deduced* from the FDs in F. We call them as inferred or implied functional dependencies.

> **Definition:** An FD $X \rightarrow Y$ is **inferred from** or **implied by** a set of dependencies F specified on R if $X \rightarrow Y$ holds in *every* legal relation state r of R; that is, whenever r satisfies all the dependencies in F, $X \rightarrow Y$ also holds in r.

In real life, it is impossible to specify all possible functional dependencies for a given situation. For example, if each department has one manager, so that Dept_no uniquely determines Mgr_ssn (Dept_no $\rightarrow$ Mgr_ssn), and a manager has a unique phone number called Mgr_phone (Mgr_ssn $\rightarrow$ Mgr_phone), then these two dependencies together imply that Dept_no $\rightarrow$ Mgr_phone. This is an inferred or implied FD and need *not* be explicitly stated in addition to the two given FDs. Therefore, it is useful to define a concept called *closure* formally that includes all possible dependencies that can be inferred from the given set F.

> **Definition.** Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the **closure** of F; it is denoted by F^+.

For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema in Figure 14.3(a):

$F = \{$Ssn $\rightarrow \{$Ename, Bdate, Address, Dnumber$\}$, Dnumber $\rightarrow \{$Dname, Dmgr_ssn$\} \}$

Some of the additional functional dependencies that we can *infer* from *F* are the following:

> Ssn → {Dname, Dmgr_ssn}
> Ssn → Ssn
> Dnumber → Dname

The closure F^+ of *F* is the set of all functional dependencies that can be inferred from *F*. To determine a systematic way to infer dependencies, we must discover a set of **inference rules** that can be used to infer new dependencies from a given set of dependencies. We consider some of these inference rules next. We use the notation $F \models X \rightarrow Y$ to denote that the functional dependency $X \rightarrow Y$ is inferred from the set of functional dependencies *F*.

In the following discussion, we use an abbreviated notation when discussing functional dependencies. We concatenate attribute variables and drop the commas for convenience. Hence, the FD $\{X,Y\} \rightarrow Z$ is abbreviated to $XY \rightarrow Z$, and the FD $\{X, Y, Z\} \rightarrow \{U, V\}$ is abbreviated to $XYZ \rightarrow UV$. We present below three rules IR1 through IR3 that are well-known inference rules for functional dependencies. They were proposed first by Armstrong (1974) and hence are known as **Armstrong's axioms**.[1]

> IR1 (reflexive rule)[2]: If $X \supseteq Y$, then $X \rightarrow Y$.
>
> IR2 (augmentation rule)[3]: $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.
>
> IR3 (transitive rule): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

Armstrong has shown that inference rules IR1 through IR3 are sound and complete. By **sound**, we mean that given a set of functional dependencies *F* specified on a relation schema *R*, any dependency that we can infer from *F* by using IR1 through IR3 holds in every relation state *r* of *R* that *satisfies the dependencies* in *F*. By **complete**, we mean that using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of *all possible dependencies* that can be inferred from *F*. In other words, the set of dependencies F^+, which we called the **closure** of *F*, can be determined from *F* by using only inference rules IR1 through IR3.

The reflexive rule (IR1) states that a set of attributes always determines itself or any of its subsets, which is obvious. Because IR1 generates dependencies that are always true, such dependencies are called *trivial*. Formally, a functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**. The augmentation rule (IR2) says that adding the same set of attributes to both the left- and right-hand sides of a dependency results in another valid dependency. According to IR3, functional dependencies are transitive.

---

[1]They are actually inference rules rather than axioms. In the strict mathematical sense, the *axioms* (given facts) are the functional dependencies in *F*, since we assume that they are correct, whereas IR1 through IR3 are the *inference rules* for inferring new functional dependencies (new facts).

[2]The reflexive rule can also be stated as $X \rightarrow X$; that is, any set of attributes functionally determines itself.

[3]The augmentation rule can also be stated as $X \rightarrow Y \models XZ \rightarrow Y$; that is, augmenting the left-hand-side attributes of an FD produces another valid FD.

Each of the preceding inference rules can be proved from the definition of functional dependency, either by direct proof or **by contradiction**. A proof by contradiction assumes that the rule does not hold and shows that this is not possible. We now prove that the first three rules IR1 through IR3 are valid. The second proof is by contradiction.

> **Proof of IR1.** Suppose that $X \supseteq Y$ and that two tuples t_1 and t_2 exist in some relation instance r of R such that $t_1 [X] = t_2 [X]$. Then $t_1[Y] = t_2[Y]$ because $X \supseteq Y$; hence, $X \to Y$ must hold in r.

> **Proof of IR2 (by contradiction).** Assume that $X \to Y$ holds in a relation instance r of R but that $XZ \to YZ$ does not hold. Then there must exist two tuples t_1 and t_2 in r such that (1) $t_1 [X] = t_2 [X]$, (2) $t_1 [Y] = t_2 [Y]$, (3) $t_1 [XZ] = t_2 [XZ]$, and (4) $t_1 [YZ] \neq t_2 [YZ]$. This is not possible because from (1) and (3) we deduce (5) $t_1 [Z] = t_2 [Z]$, and from (2) and (5) we deduce (6) $t_1 [YZ] = t_2 [YZ]$, contradicting (4).

> **Proof of IR3.** Assume that (1) $X \to Y$ and (2) $Y \to Z$ both hold in a relation r. Then for any two tuples t_1 and t_2 in r such that $t_1 [X] = t_2 [X]$, we must have (3) $t_1 [Y] = t_2 [Y]$, from assumption (1); hence we must also have (4) $t_1 [Z] = t_2 [Z]$ from (3) and assumption (2); thus $X \to Z$ must hold in r.

There are three other inference rules that follow from IR1, IR2 and IR3. They are as follows:

> IR4 (decomposition, or projective, rule): $\{X \to YZ\} \models X \to Y$.
>
> IR5 (union, or additive, rule): $\{X \to Y, X \to Z\} \models X \to YZ$.
>
> IR6 (pseudotransitive rule): $\{X \to Y, WY \to Z\} \models WX \to Z$.

The decomposition rule (IR4) says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD $X \to \{A_1, A_2, \dots, A_n\}$ into the set of dependencies $\{X \to A_1, X \to A_2, \dots, X \to A_n\}$. The union rule (IR5) allows us to do the opposite; we can combine a set of dependencies $\{X \to A_1, X \to A_2, \dots, X \to A_n\}$ into the single FD $X \to \{A_1, A_2, \dots, A_n\}$. The pseudotransitive rule (IR6) allows us to replace a set of attributes Y on the left-hand side of a dependency with another set X that functionally determines Y, and can be derived from IR2 and IR3 if we augment the first functional dependency $X \to Y$ with W (the augmentation rule) and then apply the transitive rule.

One *important cautionary note* regarding the use of these rules: Although $X \to A$ and $X \to B$ implies $X \to AB$ by the union rule stated above, $X \to A$ and $Y \to B$ does imply that $XY \to AB$. Also, $XY \to A$ does *not* necessarily imply either $X \to A$ or $Y \to A$.

Using similar proof arguments, we can prove the inference rules IR4 to IR6 and any additional valid inference rules. However, a simpler way to prove that an inference rule for functional dependencies is valid is to prove it by using inference rules that have already been shown to be valid. Thus IR4, IR5, and IR6 are regarded as a corollary of the Armstrong's basic inference rules. For example, we can prove IR4 through IR6 by using *IR1 through IR3*. We present the proof of IR5 below. Proofs of IR4 and IR6 using IR1 through IR3 are left as an exercise for the reader.

Proof of IR5 (using IR1 through IR3).

1. $X \rightarrow Y$ (given).
2. $X \rightarrow Z$ (given).
3. $X \rightarrow XY$ (using IR2 on 1 by augmenting with X; notice that $XX = X$).
4. $XY \rightarrow YZ$ (using IR2 on 2 by augmenting with Y).
5. $X \rightarrow YZ$ (using IR3 on 3 and 4).

Typically, database designers first specify the set of functional dependencies F that can easily be determined from the semantics of the attributes of R; then IR1, IR2, and IR3 are used to infer additional functional dependencies that will also hold on R. A systematic way to determine these additional functional dependencies is first to determine each set of attributes X that appears as a left-hand side of some functional dependency in F and then to determine the set of *all attributes* that are dependent on X.

Definition. For each such set of attributes X, we determine the set X^+ of attributes that are functionally determined by X based on F; X^+ is called the **closure of X under F**.

Algorithm 15.1 can be used to calculate X^+.

Algorithm 15.1. Determining X^+, the Closure of X under F

Input: A set F of FDs on a relation schema R, and a set of attributes X, which is a subset of R.

$X^+ := X$;
repeat
    old$X^+ := X^+$;
    for each functional dependency $Y \rightarrow Z$ in F do
        if $X^+ \supseteq Y$ then $X^+ := X^+ \cup Z$;
    until ($X^+ = $ oldX^+);

Algorithm 15.1 starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR3 and IR4, we add attributes to X^+, using each functional dependency in F. We keep going through all the dependencies in F (the *repeat* loop) until no more attributes are added to X^+ *during a complete cycle* (of the *for* loop) through the dependencies in F. The closure concept is useful in understanding the meaning and implications of attributes or sets of attributes in a relation. For example, consider the following relation schema about classes held at a university in a given academic year.

CLASS (Classid, Course#, Instr_name, Credit_hrs, Text, Publisher, Classroom, Capacity).
Let F, the set of functional dependencies for the above relation include the following f.d.s:
FD1: Sectionid $\rightarrow$ Course#, Instr_name, Credit_hrs, Text, Publisher, Classroom, Capacity;

FD2: Course# → Credit_hrs;
FD3: {Course#, Instr_name} → Text, Classroom;
FD4: Text → Publisher
FD5: Classroom → Capacity

Note that the above FDs express certain semantics about the data in the relation CLASS. For example, FD1 states that each class has a unique Classid. FD3 states that when a given course is offered by a certain instructor, the text is fixed and the instructor teaches that class in a fixed room. Using the inference rules about the FDs and applying the definition of closure, we can define the following closures:

{ Classid } $^+$ = { Classid , Course#, Instr_name, Credit_hrs, Text, Publisher, Classroom, Capacity } = CLASS
{ Course#} $^+$ = { Course#, Credit_hrs}
{ Course#, Instr_name } $^+$ = { Course#, Credit_hrs, Text, Publisher, Classroom, Capacity }

Note that each closure above has an interpretation that is revealing about the attribute(s) on the left-hand side. For example, the closure of Course# has only Credit_hrs besides itself. It does not include Instr_name because different instructors could teach the same course; it does not include Text because different instructors may use different texts for the same course. Note also that the closure of {Course#, Instr_nam} does not include Classid, which implies that it is not a candidate key. This further implies that a course with given Course# could be offered by different instructors, which would make the courses distinct classes.

15.1.2 Equivalence of Sets of Functional Dependencies

In this section, we discuss the equivalence of two sets of functional dependencies. First, we give some preliminary definitions.

> **Definition.** A set of functional dependencies F is said to **cover** another set of functional dependencies E if every FD in E is also in F^+; that is, if every dependency in E can be inferred from F; alternatively, we can say that E is **covered by** F.

> **Definition.** Two sets of functional dependencies E and F are **equivalent** if $E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions—E covers F *and* F covers E—hold.

We can determine whether F covers E by calculating X^+ *with respect to F* for each FD $X → Y$ *in E*, and then checking whether this X^+ includes the attributes in Y. If this is the case for *every* FD in E, then F covers E. We determine whether E and F are equivalent by checking that E covers F and F covers E. It is left to the reader as an exercise to show that the following two sets of FDs are equivalent:

$F = \{A → C, AC → D, E → AD, E → H\}$
and $G = \{A → CD, E → AH\}$

15.1.3 Minimal Sets of Functional Dependencies

Just as we applied inference rules to expand on a set *F* of FDs to arrive at F^+, its closure, it is possible to think in the opposite direction to see if we could shrink or reduce the set *F* to its *minimal form* so that the minimal set is still equivalent to the original set *F.* Informally, a **minimal cover** of a set of functional dependencies *E* is a set of functional dependencies *F* that satisfies the property that every dependency in *E* is in the closure F^+ of *F.* In addition, this property is lost if any dependency from the set *F* is removed; *F* must have no redundancies in it, and the dependencies in *F* are in a standard form.

We will use the concept of an extraneous attribute in a functional dependency for defining the minimum cover.

> **Definition:** An attribute in a functional dependency is considered an **extraneous attribute** if we can remove it without changing the closure of the set of dependencies. Formally, given *F*, the set of functional dependencies, and a functional dependency $X \rightarrow A$ in *F,* attribute *Y* is extraneous in *X* if $Y \subset X,$ and *F* logically implies $(F - (X \rightarrow A) \cup \{ (X - Y) \rightarrow A \}).$

We can formally define a set of functional dependencies *F* to be **minimal** if it satisfies the following conditions:

1. Every dependency in *F* has a single attribute for its right-hand side.
2. We cannot replace any dependency $X \rightarrow A$ in *F* with a dependency $Y \rightarrow A,$ where *Y* is a proper subset of *X,* and still have a set of dependencies that is equivalent to *F.*
3. We cannot remove any dependency from *F* and still have a set of dependencies that is equivalent to *F.*

We can think of a minimal set of dependencies as being a set of dependencies in a *standard* or *canonical form* and with *no redundancies.* Condition 1 just represents every dependency in a canonical form with a single attribute on the right-hand side, and it is a preparatory step before we can evaluate if conditions 2 and 3 are met.[4] Conditions 2 and 3 ensure that there are no redundancies in the dependencies either by having redundant attributes (referred to as extraneous attributes) on the left-hand side of a dependency (Condition 2) or by having a dependency that can be inferred from the remaining FDs in *F* (Condition 3).

> **Definition.** A **minimal cover** of a set of functional dependencies *E* is a minimal set of dependencies (in the standard canonical form[5] and without redundancy) that is equivalent to *E.* We can always find *at least one* minimal cover *F* for any set of dependencies *E* using Algorithm 15.2.

---

[4]This is a standard form to simplify the conditions and algorithms that ensure no redundancy exists in *F.* By using the inference rule IR4, we can convert a single dependency with multiple attributes on the right-hand side into a set of dependencies with single attributes on the right-hand side.

[5]It is possible to use the inference rule IR5 and combine the FDs with the same left-hand side into a single FD in the minimum cover in a nonstandard form. The resulting set is still a minimum cover, as illustrated in the example.

If several sets of FDs qualify as minimal covers of E by the definition above, it is customary to use additional criteria for *minimality*. For example, we can choose the minimal set with the *smallest number of dependencies* or with the smallest *total length* (the total length of a set of dependencies is calculated by concatenating the dependencies and treating them as one long character string).

> **Algorithm 15.2.** Finding a Minimal Cover F for a Set of Functional Dependencies E
>
> **Input:** A set of functional dependencies E.
>
> *Note*: Explanatory comments are given at the end of some of the steps. They follow the format: (*comment*).
>
> 1. Set $F := E$.
> 2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \dots , A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, \dots , X \rightarrow A_n$. (*This places the FDs in a canonical form for subsequent testing*)
> 3. For each functional dependency $X \rightarrow A$ in F
>   for each attribute B that is an element of X
>   if $\{ \{F - \{X \rightarrow A\} \} \cup \{ (X - \{B\}) \rightarrow A\} \}$ is equivalent to F
>   then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ in F.
>   (*This constitutes removal of an extraneous attribute B contained in the left-hand side X of a functional dependency $X \rightarrow A$ when possible*)
> 4. For each remaining functional dependency $X \rightarrow A$ in F
>   if $\{F - \{X \rightarrow A\} \}$ is equivalent to F,
>   then remove $X \rightarrow A$ from F. (*This constitutes removal of a redundant functional dependency $X \rightarrow A$ from F when possible*)

We illustrate the above algorithm with the following examples:

Example 1: Let the given set of FDs be E: $\{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$. We have to find the minimal cover of E.

- All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm 15.2 and can proceed to step 2. In step 2 we need to determine if $AB \rightarrow D$ has any redundant (extraneous) attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ or $A \rightarrow D$?

- Since B $\rightarrow$ A, by augmenting with B on both sides (IR2), we have $BB \rightarrow AB$, or $B \rightarrow AB$ (i). However, $AB \rightarrow D$ as given (ii).

- Hence by the transitive rule (IR3), we get from (i) and (ii), $B \rightarrow D$. Thus $AB \rightarrow D$ may be replaced by $B \rightarrow D$.

- We now have a set equivalent to original E, say E': $\{B \rightarrow A, D \rightarrow A, B \rightarrow D\}$. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.

- In step 3 we look for a redundant FD in E'. By using the transitive rule on $B \rightarrow D$ and $D \rightarrow A$, we derive $B \rightarrow A$. Hence $B \rightarrow A$ is redundant in E' and can be eliminated.

- Therefore, the minimal cover of E is F: $\{B \rightarrow D, D \rightarrow A\}$.

The reader can verify that the original set F can be inferred from E; in other words, the two sets F and E are equivalent.

Example 2: Let the given set of FDs be G: $\{A \rightarrow BCDE, CD \rightarrow E\}$.

- Here, the given FDs are NOT in the canonical form. So we first convert them into:

$$E: \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, CD \rightarrow E\}.$$

- In step 2 of the algorithm, for $CD \rightarrow E$, neither C nor D is extraneous on the left-hand side, since we cannot show that $C \rightarrow E$ or $D \rightarrow E$ from the given FDs. Hence we cannot replace it with either.

- In step 3, we want to see if any FD is redundant. Since $A \rightarrow CD$ and $CD \rightarrow E$, by transitive rule (IR3), we get $A \rightarrow E$. Thus, $A \rightarrow E$ is redundant in G.

- So we are left with the set F, equivalent to the original set G as: $\{A \rightarrow B, A \rightarrow C, A \rightarrow D, CD \rightarrow E\}$. F is the minimum cover. As we pointed out in foot-note 6, we can combine the first three FDs using the union rule (IR5) and express the minimum cover as:

$$\text{Minimum cover of } G, F: \{A \rightarrow BCD, CD \rightarrow E\}.$$

In Section 15.3, we will show algorithms that synthesize 3NF or BCNF relations from a given set of dependencies E by first finding the minimal cover F for E.

Next, we provide a simple algorithm to determine the key of a relation:

Algorithm 15.2(a). Finding a Key K for R Given a Set F of Functional Dependencies

Input: A relation R and a set of functional dependencies F on the attributes of R.

1. Set $K := R$.
2. For each attribute A in K
   {compute $(K - A)^+$ with respect to F;
   if $(K - A)^+$ contains all the attributes in R, then set $K := K - \{A\}$ };

In Algorithm 15.2(a), we start by setting K to all the attributes of R; we can say that R itself is always a **default superkey**. We then remove one attribute at a time and check whether the remaining attributes still form a superkey. Notice, too, that Algorithm 15.2(a) determines only *one key* out of the possible candidate keys for R; the key returned depends on the order in which attributes are removed from R in step 2.

15.2 Properties of Relational Decompositions

We now turn our attention to the process of decomposition that we used throughout Chapter 14 to get rid of unwanted dependencies and achieve higher normal forms. In Section 15.2.1, we give examples to show that looking at an *individual* relation to test whether it is in a higher normal form does not, on its own, guarantee a good design; rather, a *set of relations* that together form the relational database schema must possess certain additional properties to ensure a good design. In Sections 15.2.2 and 15.2.3, we discuss two of these properties: the dependency preservation property and the nonadditive (or lossless) join property. Section 15.2.4 discusses binary decompositions, and Section 15.2.5 discusses successive nonadditive join decompositions.

15.2.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present in Section 15.3 start from a single **universal relation schema** $R = \{A_1, A_2, \ldots, A_n\}$ that includes *all* the attributes of the database. We implicitly make the **universal relation assumption**, which states that every attribute name is unique. The set F of functional dependencies that should hold on the attributes of R is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas $D = \{R_1, R_2, \ldots, R_m\}$ that will become the relational database schema; D is called a **decomposition** of R.

We must make sure that each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are *lost*; formally, we have

$$\bigcup_{i=1}^{m} R_i = R$$

This is called the **attribute preservation** condition of a decomposition.

Another goal is to have each individual relation R_i in the decomposition D be in BCNF or 3NF. However, this condition is not sufficient to guarantee a good database design on its own. We must consider the decomposition of the universal relation as a whole, in addition to looking at the individual relations. To illustrate this point, consider the EMP_LOCS(Ename, Plocation) relation in Figure 14.5, which is in 3NF and also in BCNF. In fact, any relation schema with only two attributes is automatically in BCNF.[6] Although EMP_LOCS is in BCNF, it still gives rise to spurious tuples when joined with EMP_PROJ (Ssn, Pnumber, Hours, Pname, Plocation), which is not in BCNF (see the partial result of the natural join in Figure 14.6). Hence, EMP_LOCS represents a particularly bad relation schema because of its convoluted

---

[6]As an exercise, the reader should prove that this statement is true.

semantics by which Plocation gives the location of *one of the projects* on which an employee works. Joining EMP_LOCS with PROJECT(Pname, Pnumber, Plocation, Dnum) in Figure 14.2—which *is* in BCNF—using Plocation as a joining attribute also gives rise to spurious tuples. This underscores the need for other criteria that, together with the conditions of 3NF or BCNF, prevent such bad designs. In the next three subsections we discuss such additional conditions that should hold on a decomposition D as a whole.

15.2.2 Dependency Preservation Property of a Decomposition

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i. Informally, this is the *dependency preservation condition*. We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.

It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D. It is sufficient that the union of the dependencies that hold on the individual relations in D be equivalent to F. We now define these concepts more formally.

> **Definition.** Given a set of dependencies F on R, the **projection** of F on R_i, denoted by $\pi_{R_i}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i. Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all the left- and right-hand-side attributes of those dependencies are in R_i. We say that a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is, $((\pi_{R_1}(F)) \cup K \cup (\pi_{R_m}(F)))^+ = F^+$.

If a decomposition is not dependency-preserving, some dependency is **lost** in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all left- and right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN—an option that is not practical.

An example of a decomposition that does not preserve dependencies is shown in Figure 14.13(a), in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}. The decompositions in Figure 14.12, however, are dependency-preserving. Similarly, for the example in Figure 14.14, no

matter what decomposition is chosen for the relation TEACH(Student, Course, Instructor) from the three provided in the text, one or both of the dependencies originally present are bound to be lost. We now state a claim related to this property without providing any proof.

> **Claim 1.** It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation R_i in D is in 3NF.

15.2.3 Nonadditive (Lossless) Join Property of a Decomposition

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition. We already illustrated this problem in Section 14.1.4 with the example in Figures 14.5 and 14.6. Because this is a property of a decomposition of relation *schemas,* the condition of no spurious tuples should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in F. Hence, the lossless join property is always defined with respect to a specific set F of dependencies.

> **Definition.** Formally, a decomposition $D = \{R_1, R_2, \dots , R_m\}$ of R has the **lossless (nonadditive) join property** with respect to the set of dependencies F on R if, for *every* relation state r of R that satisfies F, the following holds, where $*$ is the NATURAL JOIN of all the relations in D: $*(\pi_{R_1}(r), \dots , \pi_{R_m}(r)) = r.$

The word loss in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN ($*$) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term *nonadditive join* because it describes the situation more accurately. Although the term *lossless join* has been popular in the literature, *we* used the term *nonadditive join* in describing the NJB property in Section 14.5.1. We *will henceforth use the term nonadditive join*, which is self-explanatory and unambiguous. The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term **lossy design** to refer to a design that represents a loss of information. The decomposition of EMP_PROJ(Ssn, Pnumber, Hours, Ename, Pname, Plocation) in Figure 14.3 into EMP_LOCS(Ename, Plocation) and EMP_PROJ1(Ssn, Pnumber, Hours, Pname, Plocation) in Figure 14.5 obviously does not have the nonadditive join property, as illustrated by the partial result of NATURAL JOIN in Figure 14.6. We provided a simpler test in case of binary decompositions to check if the decomposition is nonadditive—it was called the NJB property in Section 14.5.1. We provide a general procedure for testing whether any decomposition D of a relation into n relations is nonadditive with respect to a set of given functional dependencies F in the relation; it is presented as Algorithm 15.3.

Algorithm 15.3. Testing for Nonadditive Join Property

Input: A universal relation R, a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R, and a set F of functional dependencies.

Note: Explanatory comments are given at the end of some of the steps. They follow the format: (*comment*).

1. Create an initial matrix S with one row i for each relation R_i in D, and one column j for each attribute A_j in R.
2. Set $S(i, j) := b_{ij}$ for all matrix entries. (*Each b_{ij} is a distinct symbol associated with indices (i, j)*)
3. For each row i representing relation schema R_i
    {for each column j representing attribute A_j
        {if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$;};}; (*Each a_j is a distinct symbol associated with index (j)*)
4. Repeat the following loop until a *complete loop execution* results in no changes to S
    {for each functional dependency $X \rightarrow Y$ in F
        {for all rows in S *that have the same symbols* in the columns corresponding to attributes in X
            {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an a symbol for the column, set the other rows to that *same a* symbol in the column. If no a symbol exists for the attribute in any of the rows, choose one of the b symbols that appears in one of the rows for the attribute and set the other rows to that same b symbol in the column ;} ; } ;};
5. If a row is made up entirely of a symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

Given a relation R that is decomposed into a number of relations $R_1, R_2, \ldots, R_m$, Algorithm 15.3 begins the matrix S that we consider to be some relation state r of R. Row i in S represents a tuple t_i (corresponding to relation R_i) that has a symbols in the columns that correspond to the attributes of R_i and b symbols in the remaining columns. The algorithm then transforms the rows of this matrix (during the loop in step 4) so that they represent tuples that satisfy all the functional dependencies in F. At the end of step 4, any two rows in S—which represent two tuples in r—that agree in their values for the left-hand-side attributes X of a functional dependency $X \rightarrow Y$ in F will also agree in their values for the right-hand-side attributes Y. It can be shown that after applying the loop of step 4, if any row in S ends up with all a symbols, then the decomposition D has the nonadditive join property with respect to F.

If, on the other hand, no row ends up being all a symbols, D does not satisfy the lossless join property. In this case, the relation state r represented by S at the end of

the algorithm will be an example of a relation state r of R that satisfies the dependencies in F but does not satisfy the nonadditive join condition. Thus, this relation serves as a **counterexample** that proves that D does not have the nonadditive join property with respect to F. Note that the a and b symbols have no special meaning at the end of the algorithm.

Figure 15.1(a) shows how we apply Algorithm 15.3 to the decomposition of the EMP_PROJ relation schema from Figure 14.3(b)into the two relation schemas EMP_PROJ1 and EMP_LOCS in Figure 14.5(a). The loop in step 4 of the algorithm cannot change any b symbols to a symbols; hence, the resulting matrix S does not have a row with all a symbols, and so the decomposition does not have the nonadditive join property.

Figure 15.1(b) shows another decomposition of EMP_PROJ (into EMP, PROJECT, and WORKS_ON) that does have the nonadditive join property, and Figure 15.1(c) shows how we apply the algorithm to that decomposition. Once a row consists only of a symbols, we conclude that the decomposition has the nonadditive join property, and we can stop applying the functional dependencies (step 4 in the algorithm) to the matrix S.

15.2.4 Testing Binary Decompositions for the Nonadditive Join Property

Algorithm 15.3 allows us to test whether a particular decomposition D into n relations obeys the nonadditive join property with respect to a set of functional dependencies F. There is a special case of a decomposition called a **binary decomposition**—decomposition of a relation R into two relations. A test called the NJB property test, which is easier to apply than Algorithm 15.3 but is *limited* only to binary decompositions, was given in Section 14.5.1. It was used to do binary decomposition of the TEACH relation, which met 3NF but did not meet BCNF, into two relations that satisfied this property.

15.2.5 Successive Nonadditive Join Decompositions

We saw the successive decomposition of relations during the process of second and third normalization in Sections 14.3 and 14.4. To verify that these decompositions are nonadditive, we need to ensure another property, as set forth in Claim 2.

> **Claim 2 (Preservation of Nonadditivity in Successive Decompositions).** If a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition $D_i = \{Q_1, Q_2, \ldots, Q_k\}$ of R_i has the nonadditive join property with respect to the projection of F on R_i, then the decomposition $D_2 = \{R_1, R_2, \ldots, R_{i-1}, Q_1, Q_2, \ldots, Q_k, R_{i+1}, \ldots, R_m\}$ of R has the nonadditive join property with respect to F.

Figure 15.1
Nonadditive join test for *n*-ary decompositions. (a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test. (b) A decomposition of EMP_PROJ that has the lossless join property. (c) Case 2: Decomposition of EMP_PROJ into EMP, PROJECT, and WORKS_ON satisfies test.

(a) $R = \{$Ssn, Ename, Pnumber, Pname, Plocation, Hours$\}$ $\qquad D = \{R_1, R_2\}$
$R_1 = $ EMP_LOCS $ = \{$Ename, Plocation$\}$
$R_2 = $ EMP_PROJ1 $ = \{$Ssn, Pnumber, Hours, Pname, Plocation$\}$

$F = \{$Ssn $\rightarrow$ Ename; Pnumber $\rightarrow$ $\{$Pname, Plocation$\}$; $\{$Ssn, Pnumber$\}$ $\rightarrow$ Hours$\}$

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(No changes to matrix after applying functional dependencies)

(b)

EMP	
Ssn	Ename

PROJECT		
Pnumber	Pname	Plocation

WORKS_ON		
Ssn	Pnumber	Hours

(c) $R = \{$Ssn, Ename, Pnumber, Pname, Plocation, Hours$\}$ $\qquad D = \{R_1, R_2, R_3\}$
$R_1 = $ EMP $ = \{$Ssn, Ename$\}$
$R_2 = $ PROJ $ = \{$Pnumber, Pname, Plocation$\}$
$R_3 = $ WORKS_ON $ = \{$Ssn, Pnumber, Hours$\}$

$F = \{$Ssn $\rightarrow$ Ename; Pnumber $\rightarrow$ $\{$Pname, Plocation$\}$; $\{$Ssn, Pnumber$\}$ $\rightarrow$ Hours$\}$

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(Original matrix S at start of algorithm)

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	$\cancel{b_{32}}\ a_2$	a_3	$\cancel{b_{34}}\ a_4$	$\cancel{b_{35}}\ a_5$	a_6

(Matrix S after applying the first two functional dependencies; last row is all "a" symbols so we stop)

15.3 Algorithms for Relational Database Schema Design

We now give two algorithms for creating a relational decomposition from a universal relation. The first algorithm decomposes a universal relation into dependency-preserving 3NF relations that also possess the nonadditive join property. The second algorithm decomposes a universal relation schema into BCNF schemas that possess the nonadditive join property. It is not possible to design an algorithm to produce BCNF relations that satisfy both dependency preservation and nonadditive join decomposition

15.3.1 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

By now we know that it is *not possible to have all three of the following*: (1) guaranteed nonlossy (nonadditive) design, (2) guaranteed dependency preservation, and (3) all relations in BCNF. As we have stressed repeatedly, the first condition is a must and cannot be compromised. The second condition is desirable, but not a must, and may have to be relaxed if we insist on achieving BCNF. The original lost FDs can be recovered by a JOIN operation over the results of decomposition. Now we give an algorithm where we achieve conditions 1 and 2 and only guarantee 3NF. Algorithm 15.4 yields a decomposition D of R that does the following:

- Preserves dependencies
- Has the nonadditive join property
- Is such that each resulting relation schema in the decomposition is in 3NF

Algorithm 15.4 Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F (use Algorithm 15.2).
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \to A_1, X \to A_2, \dots, X \to A_k$ are the only dependencies in G with X as left-hand side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R. (Algorithm 15.2(a) may be used to find a key.)
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S.[7]

---

[7]Note that there is an additional type of dependency: R is a projection of the join of two or more relations in the schema. This type of redundancy is considered join dependency, as we discussed in Section 15.7. Hence, technically, it may continue to exist without disturbing the 3NF status for the schema.

Step 3 of Algorithm 15.4 involves identifying a key K of R. Algorithm 15.2(a) can be used to identify a key K of R based on the set of given functional dependencies F. Notice that the set of functional dependencies used to determine a key in Algorithm 15.2(a) could be either F or G, since they are equivalent.

Example 1 of Algorithm 15.4. Consider the following universal relation:

U (Emp_ssn, Pno, Esal, Ephone, Dno, Pname, Plocation)

Emp_ssn, Esal, and Ephone refer to the Social Security number, salary, and phone number of the employee. Pno, Pname, and Plocation refer to the number, name, and location of the project. Dno is the department number.

The following dependencies are present:

FD1: Emp_ssn → {Esal, Ephone, Dno}

FD2: Pno → { Pname, Plocation}

FD3: Emp_ssn, Pno → {Esal, Ephone, Dno, Pname, Plocation}

By virtue of FD3, the attribute set {Emp_ssn, Pno} represents a key of the universal relation. Hence F, the set of given FDs, includes {Emp_ssn → Esal, Ephone, Dno; Pno → Pname, Plocation; Emp_ssn, Pno → Esal, Ephone, Dno, Pname, Plocation}.

By applying the minimal cover Algorithm 15.2, in step 3 we see that Pno is an extraneous attribute in Emp_ssn, Pno → Esal, Ephone, Dno. Moreover, Emp_ssn is extraneous in Emp_ssn, Pno → Pname, Plocation. Hence the minimal cover consists of FD1 and FD2 only (FD3 being completely redundant) as follows (if we group attributes with the same left-hand side into one FD):

Minimal cover G: {Emp_ssn → Esal, Ephone, Dno; Pno → Pname, Plocation}

The second step of Algorithm 15.4 produces relations R_1 and R_2 as:

R_1 (<u>Emp_ssn</u>, Esal, Ephone, Dno)

R_2 (<u>Pno</u>, Pname, Plocation)

In step 3, we generate a relation corresponding to the key {Emp_ssn, Pno} of U. Hence, the resulting design contains:

R_1 (<u>Emp_ssn</u>, Esal, Ephone, Dno)

R_2 (<u>Pno</u>, Pname, Plocation)

R_3 (<u>Emp_ssn</u>, <u>Pno</u>)

This design achieves both the desirable properties of dependency preservation and nonadditive join.

Example 2 of Algorithm 15.4 (Case X). Consider the relation schema LOTS1A shown in Figure 14.13(a).

Assume that this relation is given as a universal relation U (Property_id, County, Lot#, Area) with the following functional dependencies:

> FD1: Property_id → Lot#, County, Area
> FD2: Lot#, County → Area, Property_id
> FD3: Area → County

These were called FD1, FD2, and FD5 in Figure 14.13(a). The meanings of the above attributes and the implication of the above functional dependencies were explained in Section 14.4. For ease of reference, let us abbreviate the above attributes with the first letter for each and represent the functional dependencies as the set

> F: { P → LCA, LC → AP, A → C }

The universal relation with abbreviated attributes is U (P, C, L, A). If we apply the minimal cover Algorithm 15.2 to F, (in step 2) we first represent the set F as

> F: {P → L, P → C, P → A, LC → A, LC → P, A → C}

In the set F, P → A can be inferred from P → LC and LC → A; hence P → A by transitivity and is therefore redundant. Thus, one possible minimal cover is

> Minimal cover GX: {P → LC, LC → AP, A → C}

In step 2 of Algorithm 15.4, we produce design X (before removing redundant relations) using the above minimal cover as

> Design X: R_1 (P̲, L, C), R_2 (L̲, C̲, A, P), and R_3 (A̲, C)

In step 4 of the algorithm, we find that R_3 is subsumed by R_2 (that is, R_3 is always a projection of R_2 and R_1 is a projection of R_2 as well). Hence both of those relations are redundant. Thus the 3NF schema that achieves both of the desirable properties is (after removing redundant relations)

> Design X: R_2 (L, C, A, P).

or, in other words it is identical to the relation LOTS1A (Property_id, Lot#, County, Area) that we had determined to be in 3NF in Section 14.4.2.

Example 2 of Algorithm 15.4 (Case *Y*). Starting with LOTS1A as the universal relation and with the same given set of functional dependencies, the second step of the minimal cover Algorithm 15.2 produces, as before,

> F: {P → C, P → A, P → L, LC → A, LC → P, A → C}

The FD LC → A may be considered redundant because LC → P and P → A implies LC → A by transitivity. Also, P → C may be considered to be redundant because P → A and A → C implies P → C by transitivity. This gives a different minimal cover as

> Minimal cover GY: { P → LA, LC → P, A → C }

The alternative design Y produced by the algorithm now is

> Design Y: S_1 (P̲, A, L), S_2 (L̲, C̲, P), and S_3 (A̲, C)

Note that this design has three 3NF relations, none of which can be considered as redundant by the condition in step 4. All FDs in the original set F are preserved. The

reader will notice that of the above three relations, relations S_1 and S_3 were produced as the BCNF design by the procedure given in Section 14.5 (implying that S_2 is redundant in the presence of S_1 and S_3). However, we cannot eliminate relation S_2 from the set of three 3NF relations above since it is not a projection of either S_1 or S_3. It is easy to see that S_2 is a valid and meaningful relation that has the two candidate keys (L, C), and P placed side-by-side. Notice further that S_2 preserves the FD LC → P, which is lost if the final design contains only S_1 and S_3. Design Y therefore remains as one possible final result of applying Algorithm 15.4 to the given universal relation that provides relations in 3NF.

The above two variations of applying Algorithm 15.4 to the same universal relation with a given set of FDs have illustrated two things:

- It is possible to generate alternate 3NF designs by starting from the same set of FDs.
- It is conceivable that in some cases the algorithm actually produces relations that satisfy BCNF and may include relations that maintain the dependency preservation property as well.

15.3.2 Nonadditive Join Decomposition into BCNF Schemas

The next algorithm decomposes a universal relation schema $R = \{A_1, A_2, \ldots, A_n\}$ into a decomposition $D = \{R_1, R_2, \ldots, R_m\}$ such that each R_i is in BCNF *and* the decomposition D has the lossless join property with respect to F. Algorithm 15.5 utilizes property NJB and claim 2 (preservation of nonadditivity in successive decompositions) to create a nonadditive join decomposition $D = \{R_1, R_2, \ldots, R_m\}$ of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in BCNF.

> **Algorithm 15.5.** Relational Decomposition into BCNF with Nonadditive Join Property
>
> **Input:** A universal relation R and a set of functional dependencies F on the attributes of R.
>
> 1. Set $D := \{R\}$;
> 2. While there is a relation schema Q in D that is not in BCNF do
>   {
>   choose a relation schema Q in D that is not in BCNF;
>   find a functional dependency $X → Y$ in Q that violates BCNF;
>   replace Q in D by two relation schemas $(Q − Y)$ and $(X \cup Y)$;
>   };

Each time through the loop in Algorithm 15.5, we decompose one relation schema Q that is not in BCNF into two relation schemas. According to property NJB for binary decompositions and claim 2, the decomposition D has the nonadditive join property. At the end of the algorithm, all relation schemas in D will be in

BCNF. We illustrated the application of this algorithm to the TEACH relation schema from Figure 14.14; it is decomposed into TEACH1(<u>Instructor</u>, <u>Student</u>) and TEACH2(<u>Instructor</u>, Course) because the dependency FD2 Instructor → Course violates BCNF.

In step 2 of Algorithm 15.5, it is necessary to determine whether a relation schema Q is in BCNF or not. One method for doing this is to test, for each functional dependency $X \rightarrow Y$ in Q, whether X^+ fails to include all the attributes in Q, thereby determining whether or not X is a (super) key in Q. Another technique is based on an observation that whenever a relation schema Q has a BCNF violation, there exists a pair of attributes A and B in Q such that $\{Q - \{A, B\}\} \rightarrow A$; by computing the closure $\{Q - \{A, B\}\}^+$ for each pair of attributes $\{A, B\}$ of Q and checking whether the closure includes A (or B), we can determine whether Q is in BCNF.

It is important to note that the theory of nonadditive join decompositions is based on the assumption that *no NULL values are allowed for the join attributes.* The next section discusses some of the problems that NULLs may cause in relational decompositions and provides a general discussion of the algorithms for relational design by synthesis presented in this section.

15.4 About Nulls, Dangling Tuples, and Alternative Relational Designs

In this section, we discuss a few general issues related to problems that arise when relational design is not approached properly.

15.4.1 Problems with NULL Values and Dangling Tuples

We must carefully consider the problems associated with NULLs when designing a relational database schema. There is no fully satisfactory relational design theory as yet that includes NULL values. One problem occurs when some tuples have NULL values for attributes that will be used to join individual relations in the decomposition. To illustrate this, consider the database shown in Figure 15.2(a), where two relations EMPLOYEE and DEPARTMENT are shown. The last two employee tuples—'Berger' and 'Benitez'—represent newly hired employees who have not yet been assigned to a department (assume that this does not violate any integrity constraints). Now suppose that we want to retrieve a list of (Ename, Dname) values for all the employees. If we apply the NATURAL JOIN operation on EMPLOYEE and DEPARTMENT (Figure 15.2(b)), the two aforementioned tuples will *not* appear in the result. The OUTER JOIN operation, discussed in Chapter 8, can deal with this problem. Recall that if we take the LEFT OUTER JOIN of EMPLOYEE with DEPARTMENT, tuples in EMPLOYEE that have NULL for the join attribute will still appear in the result, joined with an *imaginary* tuple in DEPARTMENT that has NULLs for all its attribute values. Figure 15.2(c) shows the result.

In general, whenever a relational database schema is designed in which two or more relations are interrelated via foreign keys, particular care must be devoted to

watching for potential NULL values in foreign keys. This can cause unexpected loss of information in queries that involve joins on that foreign key. Moreover, if NULLs occur in other attributes, such as Salary, their effect on built-in functions such as SUM and AVERAGE must be carefully evaluated.

A related problem is that of *dangling tuples*, which may occur if we carry a decomposition too far. Suppose that we decompose the EMPLOYEE relation in Figure 15.2(a) further into EMPLOYEE_1 and EMPLOYEE_2, shown in Figures 15.3(a) and 15.3(b). If we apply the NATURAL JOIN operation to EMPLOYEE_1 and EMPLOYEE_2, we get the original EMPLOYEE relation. However, we may use the alternative representation, shown in Figure 15.3(c), where we *do not include a tuple* in EMPLOYEE_3 if the employee has not been assigned a department (instead of including a tuple with NULL for Dnum as in EMPLOYEE_2). If we use EMPLOYEE_3 instead of EMPLOYEE_2 and apply a NATURAL JOIN on EMPLOYEE_1 and EMPLOYEE_3, the tuples for Berger and Benitez will not appear in the result; these are called **dangling tuples** in EMPLOYEE_1 because they are represented in only one of the two relations that represent employees, and hence they are lost if we apply an (INNER) JOIN operation.

15.4.2 Discussion of Normalization Algorithms and Alternative Relational Designs

One of the problems with the normalization algorithms we described is that the database designer must first specify *all* the relevant functional dependencies among the database attributes. This is not a simple task for a large database with hundreds of attributes. Failure to specify one or two important dependencies may result in an undesirable design. Another problem is that these algorithms are *not deterministic* in general. For example, the *synthesis algorithms* (Algorithms 15.4 and 15.5) require the specification of a minimal cover G for the set of functional dependencies F. Because there may be, in general, many minimal covers corresponding to F, as we illustrated in Example 2 of Algorithm 15.4 above, the algorithm can give different designs depending on the particular minimal cover used. Some of these designs may not be desirable. The decomposition algorithm to achieve BCNF (Algorithm 15.5) depends on the order in which the functional dependencies are supplied to the algorithm to check for BCNF violation. Again, it is possible that many different designs may arise. Some of the designs may be preferred, whereas others may be undesirable.

It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF (instead of 3NF, as in Algorithm 15.4). We can check the 3NF relation schemas in the decomposition individually to see whether each satisfies BCNF. If some relation schema R_i is not in BCNF, we can choose to decompose it further or to leave it as it is in 3NF (with some possible update anomalies). We showed by using the bottom-up approach to design that different minimal covers in cases X and Y of Example 2 under Algorithm 15.4 produced different sets of relations

(a)

EMPLOYEE

Ename	Ssn	Bdate	Address	Dnum
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	NULL

Figure 15.2
Issues with NULL-value joins. (a) Some EMPLOYEE tuples have NULL for the join attribute Dnum. (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations. (c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

DEPARTMENT

Dname	Dnum	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

(b)

Ename	Ssn	Bdate	Address	Dnum	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

(c)

Ename	Ssn	Bdate	Address	Dnum	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL	NULL	NULL
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX	NULL	NULL	NULL

(a) EMPLOYEE_1

Ename	Ssn	Bdate	Address
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX

(b) EMPLOYEE_2

Ssn	Dnum
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	NULL
888664444	NULL

(c) EMPLOYEE_3

Ssn	Dnum
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

Figure 15.3
The dangling tuple problem.
(a) The relation EMPLOYEE_1 (includes all attributes of EMPLOYEE from Figure 15.2(a) except Dnum).
(b) The relation EMPLOYEE_2 (includes Dnum attribute with NULL values).
(c) The relation EMPLOYEE_3 (includes Dnum attribute but does not include tuples for which Dnum has NULL values).

based on minimal cover. The design X produced the 3NF design as LOTS1A (Property_id, County, Lot#, Area) relation, which is in 3NF but not BCNF. Alternately, design Y produced three relations: S_1 (Property_id, Area, Lot#), S_2 (Lot#, County, Property_id), and S_3 (Area, County). If we test each of these three relations, we find that they are in BCNF. We also saw previously that if we apply Algorithm 15.5 to LOTS1Y to decompose it into BCNF relations, the resulting design contains only S_1 and S_3 as a BCNF design. In summary, the above examples of cases (called Case X and Case Y) driven by different minimum covers for the same universal schema amply illustrate that alternate designs will result by the application of the bottom-up design algorithms we presented in Section 15.3.

Table 15.1 summarizes the properties of the algorithms discussed in this chapter so far.

Table 15.1   Summary of the Algorithms Discussed in This Chapter

Algorithm	Input	Output	Properties/Purpose	Remarks
15.1	An attribute or a set of attributes X, and a set of FDs F	A set of attributes in the closure of X with respect to F	Determine all the attributes that can be functionally determined from X	The closure of a key is the entire relation
15.2	A set of functional dependencies F	The minimal cover of functional dependencies	To determine the minimal cover of a set of dependencies F	Multiple minimal covers may exist— depends on the order of selecting functional dependencies
15.2a	Relation schema R with a set of functional dependencies F	Key K of R	To find a key K (that is a subset of R)	The entire relation R is always a default superkey
15.3	A decomposition D of R and a set F of functional dependencies	Boolean result: yes or no for nonadditive join property	Testing for nonadditive join decomposition	See a simpler test NJB in Section 14.5 for binary decompositions
15.4	A relation R and a set of functional dependencies F	A set of relations in 3NF	Nonadditive join and dependency-preserving decomposition	May not achieve BCNF, but achieves *all* desirable properties and 3NF
15.5	A relation R and a set of functional dependencies F	A set of relations in BCNF	Nonadditive join decomposition	No guarantee of dependency preservation
15.6	A relation R and a set of functional and multivalued dependencies	A set of relations in 4NF	Nonadditive join decomposition	No guarantee of dependency preservation

15.5 Further Discussion of Multivalued Dependencies and 4NF

We introduced and defined the concept of multivalued dependencies and used it to define the fourth normal form in Section 14.6. In this section, we discuss MVDs to make our treatment complete by stating the rules of inference with MVDs.

15.5.1 Inference Rules for Functional and Multivalued Dependencies

As with functional dependencies (FDs), inference rules for MVDs have been developed. It is better, though, to develop a unified framework that includes both FDs and MVDs so that both types of constraints can be considered together. The

following inference rules IR1 through IR8 form a sound and complete set for inferring functional and multivalued dependencies from a given set of dependencies. Assume that all attributes are included in a *universal* relation schema $R = \{A_1, A_2, \ldots, A_n\}$ and that X, Y, Z, and W are subsets of R.

IR1 (reflexive rule for FDs): If $X \supseteq Y$, then $X \rightarrow Y$.

IR2 (augmentation rule for FDs): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

IR3 (transitive rule for FDs): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

IR4 (complementation rule for MVDs): $\{X \twoheadrightarrow R\} \models \{X \twoheadrightarrow (R - (X \cup))\}$.

IR5 (augmentation rule for MVDs): If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.

IR6 (transitive rule for MVDs): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (X - Y)$.

IR7 (replication rule for FD to MVD): $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.

IR8 (coalescence rule for FDs and MVDs): If $X \twoheadrightarrow Y$ and there exists W with the properties that (a) $W \cap Y$ is empty, (b) $W \rightarrow Z$, and (c) $Y \supseteq Z$, then $X \rightarrow Z$.

IR1 through IR3 are Armstrong's inference rules for FDs alone. IR4 through IR6 are inference rules pertaining to MVDs only. IR7 and IR8 relate FDs and MVDs. In particular, IR7 says that a functional dependency is a *special case* of a multivalued dependency; that is, every FD is also an MVD because it satisfies the formal definition of an MVD. However, this equivalence has a catch: An FD $X \rightarrow Y$ is an MVD $X \twoheadrightarrow Y$ with the *additional implicit restriction* that at most one value of Y is associated with each value of X.[8] Given a set F of functional and multivalued dependencies specified on $R = \{A_1, A_2, \ldots, A_n\}$, we can use IR1 through IR8 to infer the (complete) set of all dependencies (functional or multivalued) F^+ that will hold in every relation state r of R that satisfies F. We again call F^+ the **closure** of F.

15.5.2 Fourth Normal Form Revisited

We restate the definition of **fourth normal form** (**4NF**) from Section 14.6:

Definition. A relation schema R is in 4**NF** with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in F^+, X in F^+, X is a superkey for R.

To illustrate the importance of 4NF, Figure 15.4(a) shows the EMP relation in Figure 14.15 with an additional employee, 'Brown', who has three dependents ('Jim', 'Joan', and 'Bob') and works on four different projects ('W', 'X', 'Y', and 'Z'). There are 16 tuples in EMP in Figure 15.4(a). If we decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, as shown in Figure 15.4(b), we need to store a total of only 11 tuples in both relations. Not only would the decomposition save on storage, but the update anomalies associated with multivalued dependencies would also be avoided. For example, if 'Brown' starts working on a new additional project 'P', we must insert *three* tuples in EMP—one for each dependent. If we forget to

---

[8]That is, the set of values of Y determined by a value of X is restricted to being a singleton set with only one value. Hence, in practice, we never view an FD as an MVD.

(a) **EMP**

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) **EMP_PROJECTS**

Ename	Pname
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

Ename	Dname
Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

Figure 15.4
Decomposing a relation state of EMP that is not in 4NF. (a) EMP relation with additional tuples. (b) Two corresponding 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

insert any one of those, the relation violates the MVD and becomes inconsistent in that it incorrectly implies a relationship between project and dependent.

If the relation has nontrivial MVDs, then insert, delete, and update operations on single tuples may cause additional tuples to be modified besides the one in question. If the update is handled incorrectly, the meaning of the relation may change. However, after normalization into 4NF, these update anomalies disappear. For example, to add the information that 'Brown' will be assigned to project 'P', only a single tuple need be inserted in the 4NF relation EMP_PROJECTS.

The EMP relation in Figure 14.15(a) is not in 4NF because it represents two *independent* 1:N relationships—one between employees and the projects they work on and the other between employees and their dependents. We sometimes have a relationship among three entities that is a legitimate three-way relationship and not a combination of two binary relationships among three participating entities, such as the SUPPLY relation shown in Figure 14.15(c). (Consider only the tuples in Figure 14.5(c) *above* the dashed line for now.) In this case a tuple represents a supplier supplying a specific part *to a particular project,* so there are *no* nontrivial MVDs. Hence, the SUPPLY all-key relation is already in 4NF and should not be decomposed.

15.5.3 Nonadditive Join Decomposition into 4NF Relations

Whenever we decompose a relation schema R into $R_1 = (X \cup Y)$ and $R_2 = (R - Y)$ based on an MVD $X \twoheadrightarrow Y$ that holds in R, the decomposition has the nonadditive join property. It can be shown that this is a necessary and sufficient condition for decomposing a schema into two schemas that have the nonadditive join property, as given by Property NJB′ that is a further generalization of Property NJB given earlier in Section 14.5.1. Property NJB dealt with FDs only, whereas NJB′ deals with both FDs and MVDs (recall that an FD is also an MVD).

> **Property NJB′.** The relation schemas R_1 and R_2 form a nonadditive join decomposition of R with respect to a set F of functional *and* multivalued dependencies if and only if
>
> $$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$$
>
> or, by symmetry, if and only if
>
> $$(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$$

We can use a slight modification of Algorithm 15.5 to develop Algorithm 15.7, which creates a nonadditive join decomposition into relation schemas that are in 4NF (rather than in BCNF). As with Algorithm 15.5, Algorithm 15.7 does *not* necessarily produce a decomposition that preserves FDs.

> **Algorithm 15.7.** Relational Decomposition into 4NF Relations with Nonadditive Join Property
>
> **Input:** A universal relation R and a set of functional and multivalued dependencies F
>
> 1. Set $D := \{ R \}$;
> 2. While there is a relation schema Q in D that is not in 4NF, do
>   { choose a relation schema Q in D that is not in 4NF;
>   find a nontrivial MVD $X \twoheadrightarrow Y$ in Q that violates 4NF;
>   replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
>   };

15.6 Other Dependencies and Normal Forms

15.6.1 Join Dependencies and the Fifth Normal Form

We already introduced another type of dependency called join dependency (JD) in Section 14.7. It arises when a relation is decomposable into a set of projected relations that can be joined back to yield the original relation. After defining JD, we defined the fifth normal form based on it in Section 14.7. Fifth normal form has also been known as project join normal form or PJNF (Fagin, 1979). A practical problem with this and some additional dependencies (and related normal forms such as DKNF, which is defined in Section 15.6.3) is that they are difficult to discover.

Furthermore, there are no sets of sound and complete inference rules to reason about them. In the remaining part of this section, we introduce some other types of dependencies that have been identified. Among them, the inclusion dependencies and those based on arithmetic or similar functions are used frequently.

15.6.2 Inclusion Dependencies

Inclusion dependencies were defined in order to formalize two types of interrelational constraints:

- The foreign key (or referential integrity) constraint cannot be specified as a functional or multivalued dependency because it relates attributes across relations.
- The constraint between two relations that represent a class/subclass relationship (see Chapters 4 and 9) also has no formal definition in terms of the functional, multivalued, and join dependencies.

Definition. An **inclusion dependency** $R.X < S.Y$ between two sets of attributes—X of relation schema R, and Y of relation schema S—specifies the constraint that, at any specific time when r is a relation state of R and s is a relation state of S, we must have

$$\pi_X(r(R)) \subseteq \pi_Y(s(S))$$

The $\subseteq$ (subset) relationship does not necessarily have to be a proper subset. Obviously, the sets of attributes on which the inclusion dependency is specified—X of R and Y of S—must have the same number of attributes. In addition, the domains for each pair of corresponding attributes should be compatible. For example, if $X = \{A_1, A_2, \ldots, A_n\}$ and $Y = \{B_1, B_2, \ldots, B_n\}$, one possible correspondence is to have dom(A_i) *compatible with* dom(B_i) for $1 \leq i \leq n$. In this case, we say that A_i **corresponds to** B_i.

For example, we can specify the following inclusion dependencies on the relational schema in Figure 14.1:

DEPARTMENT.Dmgr_ssn < EMPLOYEE.Ssn

WORKS_ON.Ssn < EMPLOYEE.Ssn

EMPLOYEE.Dnumber < DEPARTMENT.Dnumber

PROJECT.Dnum < DEPARTMENT.Dnumber

WORKS_ON.Pnumber < PROJECT.Pnumber

DEPT_LOCATIONS.Dnumber < DEPARTMENT.Dnumber

All the preceding inclusion dependencies represent **referential integrity constraints**. We can also use inclusion dependencies to represent **class/subclass relationships**. For example, in the relational schema of Figure 9.6, we can specify the following inclusion dependencies:

EMPLOYEE.Ssn < PERSON.Ssn

ALUMNUS.Ssn < PERSON.Ssn

STUDENT.Ssn < PERSON.Ssn

As with other types of dependencies, there are *inclusion dependency inference rules* (IDIRs). The following are three examples:

IDIR1 (reflexivity): $R.X < R.X$.

IDIR2 (attribute correspondence): If $R.X < S.Y$, where $X = \{A_1, A_2, \ldots, A_n\}$ and $Y = \{B_1, B_2, \ldots, B_n\}$ and A_i *corresponds to* B_i, then $R.A_i < S.B_i$ for $1 \leq i \leq n$.

IDIR3 (transitivity): If $R.X < S.Y$ and $S.Y < T.Z$, then $R.X < T.Z$.

The preceding inference rules were shown to be sound and complete for inclusion dependencies. So far, no normal forms have been developed based on inclusion dependencies.

15.6.3 Functional Dependencies Based on Arithmetic Functions and Procedures

Sometimes some attributes in a relation may be related via some arithmetic function or a more complicated functional relationship. As long as a unique value of Y is associated with every X, we can still consider that the FD $X \rightarrow Y$ exists. For example, in the relation

ORDER_LINE (Order#, Item#, Quantity, Unit_price, Extended_price, Discounted_price)

each tuple represents an item from an order with a particular quantity, and the price per unit for that item. In this relation, (Quantity, Unit_price) $\rightarrow$ Extended_price by the formula

Extended_price = Unit_price * Quantity

Hence, there is a unique value for Extended_price for every pair (Quantity, Unit_price), and thus it conforms to the definition of functional dependency.

Moreover, there may be a procedure that takes into account the quantity discounts, the type of item, and so on and computes a discounted price for the total quantity ordered for that item. Therefore, we can say

(Item#, Quantity, Unit_price) $\rightarrow$ Discounted_price, or
(Item#, Quantity, Extended_price) $\rightarrow$ Discounted_price

To check the above FDs, a more complex procedure COMPUTE_TOTAL_PRICE may have to be called into play. Although the above kinds of FDs are technically present in most relations, they are not given particular attention during normalization. They may be relevant during the loading of relations and during query processing because populating or retrieving the attribute on the right-hand side of the dependency requires the execution of a procedure such as the one mentioned above.

15.6.4 Domain-Key Normal Form

There is no hard-and-fast rule about defining normal forms only up to 5NF. Historically, the process of normalization and the process of discovering undesirable

dependencies were carried through 5NF, but it has been possible to define stricter normal forms that take into account additional types of dependencies and constraints. The idea behind **domain-key normal form** (**DKNF**) is to specify (theoretically, at least) the *ultimate normal form* that takes into account all possible types of dependencies and constraints. A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation. For a relation in DKNF, it becomes straightforward to enforce all database constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.

However, because of the difficulty of including complex constraints in a DKNF relation, its practical utility is limited, since it may be quite difficult to specify general integrity constraints. For example, consider a relation CAR(Make, Vin#) (where Vin# is the vehicle identification number) and another relation MANUFACTURE(Vin#, Country) (where Country is the country of manufacture). A general constraint may be of the following form: *If the Make is either 'Toyota' or 'Lexus', then the first character of the Vin# is a 'J' if the country of manufacture is 'Japan'; if the Make is 'Honda' or 'Acura', the second character of the Vin# is a 'J' if the country of manufacture is 'Japan'.* There is no simplified way to represent such constraints short of writing a procedure (or general assertions) to test them. The procedure COMPUTE_TOTAL_PRICE above is an example of such procedures needed to enforce an appropriate integrity constraint.

For these reasons, although the concept of DKNF is appealing and appears straightforward, it cannot be directly tested or implemented with any guarantees of consistency or non-redundancy of design. Hence it is not used much in practice.

15.7 Summary

In this chapter we presented a further set of topics related to dependencies, a discussion of decomposition, and several algorithms related to them as well as to the process of designing 3NF, BCNF, and 4NF relations from a given set of functional dependencies and multivalued dependencies. In Section 15.1 we presented inference rules for functional dependencies (FDs), the notion of closure of an attribute, the notion of closure of a set of functional dependencies, equivalence among sets of functional dependencies, and algorithms for finding the closure of an attribute (Algorithm 15.1) and the minimal cover of a set of FDs (Algorithm 15.2). We then discussed two important properties of decompositions: the nonadditive join property and the dependency-preserving property. An algorithm to test for an *n*-way nonadditive decomposition of a relation (Algorithm 15.3) was presented. A simpler test for checking for nonadditive binary decompositions (property NJB) has already been described in Section 14.5.1. We then discussed relational design by synthesis, based on a set of given functional dependencies. The *relational synthesis algorithm* (Algorithm 15.4) creates 3NF relations from a universal relation schema based on a given set of functional dependencies that has been specified by

the database designer. The *relational decomposition algorithms* (such as Algorithms 15.5 and 15.6) create BCNF (or 4NF) relations by successive nonadditive decomposition of unnormalized relations into two component relations at a time. We saw that it is possible to synthesize 3NF relation schemas that meet both of the above properties; however, in the case of BCNF, it is possible to aim only for the nonadditiveness of joins—dependency preservation *cannot* be necessarily guaranteed. If the designer has to aim for one of these two, the nonadditive join condition is an absolute must. In Section 15.4 we showed how certain difficulties arise in a collection of relations due to null values that may exist in relations in spite of the relations being individually in 3NF or BCNF. Sometimes when decomposition is improperly carried too far, certain "dangling tuples" may result that do not participate in results of joins and hence may become invisible. We also showed how algorithms such as 15.4 for 3NF synthesis could lead to alternative designs based on the choice of minimum cover. We revisited multivalued dependencies (MVDs) in Section 15.5. MVDs arise from an improper combination of two or more independent multivalued attributes in the same relation, and MVDs result in a combinational expansion of the tuples used to define fourth normal form (4NF). We discussed inference rules applicable to MVDs and discussed the importance of 4NF. Finally, in Section 15.6 we discussed inclusion dependencies, which are used to specify referential integrity and class/subclass constraints, and pointed out the need for arithmetic functions or more complex procedures to enforce certain functional dependency constraints. We concluded with a brief discussion of the domain-key normal form (DKNF).

Review Questions

15.1. What is the role of Armstrong's inference rules (inference rules IR1 through IR3) in the development of the theory of relational design?

15.2. What is meant by the completeness and soundness of Armstrong's inference rules?

15.3. What is meant by the closure of a set of functional dependencies? Illustrate with an example.

15.4. When are two sets of functional dependencies equivalent? How can we determine their equivalence?

15.5. What is a minimal set of functional dependencies? Does every set of dependencies have a minimal equivalent set? Is it always unique?

15.6. What is meant by the attribute preservation condition on a decomposition?

15.7. Why are normal forms alone insufficient as a condition for a good schema design?

15.8. What is the dependency preservation property for a decomposition? Why is it important?

15.9. Why can we *not* guarantee that BCNF relation schemas will be produced by dependency-preserving decompositions of non-BCNF relation schemas? Give a counterexample to illustrate this point.

15.10. What is the lossless (or nonadditive) join property of a decomposition? Why is it important?

15.11. Between the properties of dependency preservation and losslessness, which one must definitely be satisfied? Why?

15.12. Discuss the NULL value and dangling tuple problems.

15.13. Illustrate how the process of creating first normal form relations may lead to multivalued dependencies. How should the first normalization be done properly so that MVDs are avoided?

15.14. What types of constraints are inclusion dependencies meant to represent?

15.15. How do template dependencies differ from the other types of dependencies we discussed?

15.16. Why is the domain-key normal form (DKNF) known as the ultimate normal form?

Exercises

15.17. Show that the relation schemas produced by Algorithm 15.4 are in 3NF.

15.18. Show that, if the matrix S resulting from Algorithm 15.3 does not have a row that is all a symbols, projecting S on the decomposition and joining it back will always produce at least one spurious tuple.

15.19. Show that the relation schemas produced by Algorithm 15.5 are in BCNF.

15.20. Write programs that implement Algorithms 15.4 and 15.5.

15.21. Consider the relation REFRIG(Model#, Year, Price, Manuf_plant, Color), which is abbreviated as REFRIG(M, Y, P, MP, C), and the following set F of functional dependencies: $F = \{M \rightarrow MP, \{M, Y\} \rightarrow P, MP \rightarrow C\}$

    a. Evaluate each of the following as a candidate key for REFRIG, giving reasons why it can or cannot be a key: {M}, {M, Y}, {M, C}.

    b. Based on the above key determination, state whether the relation REFRIG is in 3NF and in BCNF, and provide proper reasons.

    c. Consider the decomposition of REFRIG into $D = \{R_1(M, Y, P), R_2(M, MP, C)\}$. Is this decomposition lossless? Show why. (You may consult the test under Property NJB in Section 14.5.1.)

15.22. Specify all the inclusion dependencies for the relational schema in Figure 5.5.

15.23. Prove that a functional dependency satisfies the formal definition of multivalued dependency.

15.24. Consider the example of normalizing the LOTS relation in Sections 14.4 and 14.5. Determine whether the decomposition of LOTS into {LOTS1AX, LOTS1AY, LOTS1B, LOTS2} has the lossless join property by applying Algorithm 15.3 and also by using the test under property NJB from Section 14.5.1.

15.25. Show how the MVDs Ename $\twoheadrightarrow$ and Ename $\twoheadrightarrow$ Dname in Figure 14.15(a) may arise during normalization into 1NF of a relation, where the attributes Pname and Dname are multivalued.

15.26. Apply Algorithm 15.2(a) to the relation in Exercise 14.24 to determine a key for R. Create a minimal set of dependencies G that is equivalent to F, and apply the synthesis algorithm (Algorithm 15.4) to decompose R into 3NF relations.

15.27. Repeat Exercise 15.26 for the functional dependencies in Exercise 14.25.

15.28. Apply the decomposition algorithm (Algorithm 15.5) to the relation R and the set of dependencies F in Exercise 15.24. Repeat for the dependencies G in Exercise 15.25.

15.29. Apply Algorithm 15.2(a) to the relations in Exercises 14.27 and 14.28 to determine a key for R. Apply the synthesis algorithm (Algorithm 15.4) to decompose R into 3NF relations and the decomposition algorithm (Algorithm 15.5) to decompose R into BCNF relations.

15.31. Consider the following decompositions for the relation schema R of Exercise 14.24. Determine whether each decomposition has (1) the dependency preservation property, and (2) the lossless join property, with respect to F. Also determine which normal form each relation in the decomposition is in.

a. $D_1 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C\}$, $R_2 = \{A, D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

b. $D_2 = \{R_1, R_2, R_3\}$; $R_1 = \{A, B, C, D, E\}$, $R_2 = \{B, F, G, H\}$, $R_3 = \{D, I, J\}$

c. $D_3 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C, D\}$, $R_2 = \{D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$

Laboratory Exercises

Note: These exercises use the DBD (Data Base Designer) system that is described in the laboratory manual. The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for Problem 14.24 are coded as:

```
R = [a, b, c, d, e, f, g, h, i, j]
F = [[[a, b],[c]],
     [[a],[d, e]],
     [[b],[f]],
     [[f],[g, h]],
     [[d],[i, j]]]
```

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

15.33. Using the DBD system, verify your answers to the following exercises:

   a. 15.24
   b. 15.26
   c. 15.27
   d. 15.28
   e. 15.29
   f. 15.31 (a) and (b)
   g. 15.32 (a) and (c)

Selected Bibliography

The books by Maier (1983) and Atzeni and De Antonellis (1993) include a comprehensive discussion of relational dependency theory. Algorithm 15.4 is based on the normalization algorithm presented in Biskup et al. (1979). The decomposition algorithm (Algorithm 15.5) is due to Bernstein (1976). Tsou and Fischer (1982) give a polynomial-time algorithm for BCNF decomposition.

The theory of dependency preservation and lossless joins is given in Ullman (1988), where proofs of some of the algorithms discussed here appear. The lossless join property is analyzed in Aho et al. (1979). Algorithms to determine the keys of a relation from functional dependencies are given in Osborn (1977); testing for BCNF is discussed in Osborn (1979). Testing for 3NF is discussed in Tsou and Fischer (1982). Algorithms for designing BCNF relations are given in Wang (1990) and Hernandez and Chan (1991).

Multivalued dependencies and fourth normal form are defined in Zaniolo (1976) and Nicolas (1978). Many of the advanced normal forms are due to Fagin: the fourth normal form in Fagin (1977), PJNF in Fagin (1979), and DKNF in Fagin (1981). The set of sound and complete rules for functional and multivalued dependencies was given by Beeri et al. (1977). Join dependencies are discussed by Rissanen (1977) and Aho et al. (1979). Inference rules for join dependencies are given by Sciore (1982). Inclusion dependencies are discussed by Casanova et al. (1981) and analyzed further in Cosmadakis et al. (1990). Their use in optimizing relational schemas is discussed in Casanova et al. (1989). Template dependencies, which are a general form of dependencies based on hypotheses and conclusion tuples, are discussed by Sadri and Ullman (1982). Other dependencies are discussed in Nicolas (1978), Furtado (1978), and Mendelzon and Maier (1979). Abiteboul et al. (1995) provides a theoretical treatment of many of the ideas presented in this chapter and Chapter 14.

This page intentionally left blank

part **7**

File Structures, Hashing, Indexing, and Physical Database Design

This page intentionally left blank

Disk Storage, Basic File Structures, Hashing, and Modern Storage Architectures

Databases are stored physically as files of records, which are typically stored on magnetic disks. This chapter and the next deal with the organization of databases in storage and the techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called *indexes*. These structures are often referred to as **physical database file structures** and are at the physical level of the three-schema architecture described in Chapter 2. We start in Section 16.1 by introducing the concepts of computer storage hierarchies and how they are used in database systems. Section 16.2 is devoted to a description of magnetic disk storage devices and their characteristics, flash memory, and solid-state drives and optical drives and magnetic tape storage devices used for archiving data. We also discuss techniques for making access from disks more efficient. After discussing different storage technologies, we turn our attention to the methods for physically organizing data on disks. Section 16.3 covers the technique of double buffering, which is used to speed retrieval of multiple disk blocks. We also discuss buffer management and buffer replacement strategies. In Section 16.4 we discuss various ways of formatting and storing file records on disk. Section 16.5 discusses the various types of operations that are typically applied to file records. We present three primary methods for organizing file records on disk: unordered records, in Section 16.6; ordered records, in Section 16.7; and hashed records, in Section 16.8.

Section 16.9 briefly introduces files of mixed records and other primary methods for organizing records, such as B-trees. These are particularly relevant for storage of object-oriented databases, which we discussed in Chapter 11. Section 16.10

describes RAID (redundant arrays of inexpensive (or independent) disks)—a data storage system architecture that is commonly used in large organizations for better reliability and performance. Finally, in Section 16.11 we describe modern developments in the storage architectures that are important for storing enterprise data: storage area networks (SANs), network-attached storage (NAS), iSCSI (Internet SCSI—small computer system interface), and other network-based storage protocols, which make storage area networks more affordable without the use of the Fibre Channel infrastructure and hence are becoming widely accepted in industry. We also discuss storage tiering and object-based storage. Section 16.12 summarizes the chapter. In Chapter 17 we discuss techniques for creating auxiliary data structures, called indexes, which speed up the search for and retrieval of records. These techniques involve storage of auxiliary data, called index files, in addition to the file records themselves.

Chapters 16 and 17 may be browsed through or even omitted by readers who have already studied file organizations and indexing in a separate course. The material covered here, in particular Sections 16.1 through 16.8, is necessary for understanding Chapters 18 and 19, which deal with query processing and optimization, as well as database tuning for improving performance of queries.

16.1 Introduction

The collection of data that makes up a computerized database must be stored physically on some computer **storage medium**. The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a *storage hierarchy* that includes two main categories:

- **Primary storage.** This category includes storage media that can be operated on directly by the computer's *central processing unit* (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity. Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than demanded by typical enterprise-level databases. The contents of main memory are lost in case of power failure or a system crash.
- **Secondary storage.** The primary choice of storage medium for online storage of enterprise databases has been magnetic disks. However, flash memories are becoming a common medium of choice for storing moderate amounts of permanent data. When used as a substitute for a disk drive, such memory is called a **solid-state drive** (SSD).
- **Tertiary storage.** Optical disks (CD-ROMs, DVDs, and other similar storage media) and tapes are removable media used in today's systems as offline storage for archiving databases and hence come under the category called tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

We first give an overview of the various storage devices used for primary, secondary, and tertiary storage in Section 16.1.1, and in Section 16.1.2 we discuss how databases are typically handled in the storage hierarchy.

16.1.1 Memory Hierarchies and Storage Devices[1]

In a modern computer system, data resides and is transported throughout a hierarchy of storage media. The highest-speed memory is the most expensive and is therefore available with the least capacity. The lowest-speed memory is offline tape storage, which is essentially available in indefinite storage capacity.

At the *primary storage level,* the memory hierarchy includes, at the most expensive end, **cache memory**, which is a static RAM (random access memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining. The next level of primary storage is DRAM (dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called **main memory**. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility[2] and lower speed compared with static RAM.

At the *secondary and tertiary storage level,* the hierarchy includes magnetic disks; **mass storage** in the form of CD-ROM (compact disk–read-only memory) and DVD (digital video disk or digital versatile disk) devices; and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1,000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1,000 GB). The word *petabyte* (1,000 terabytes or 10**15 bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Programs reside and execute in dynamic random-access memory (DRAM). Generally, large permanent databases reside on secondary storage (magnetic disks), and portions of the database are read into and written from buffers in main memory as needed. Nowadays, personal computers and workstations have large main memories of hundreds of megabytes of RAM and DRAM, so it is becoming possible to load a large part of the database into main memory. Eight to sixteen GB of main memory is becoming commonplace on laptops, and servers with 256 GB capacity are not uncommon. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), which results in **main memory databases**; these are particularly useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.

Flash Memory. Between DRAM and magnetic disk storage, another form of memory, **flash memory**, is becoming common, particularly because it is nonvolatile.

---

[1]The authors appreciate the valuable input of Dan Forsyth regarding the current status of storage systems in enterprises. The authors also wish to thank Satish Damle for his suggestions.

[2]Volatile memory typically loses its contents in case of a power outage, whereas nonvolatile memory does not.

Flash memories are high-density, high-performance memories using EEPROM (electrically erasable programmable read-only memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memories come in two types called NAND and NOR flash based on the type of logic circuits used. The NAND flash devices have a higher storage capacity for a given cost and are used as the data storage medium in appliances with capacities ranging from 8 GB to 64 GB for the popular cards that cost less than a dollar per GB. Flash devices are used in cameras, MP3/MP4 players, cell phones, PDAs (personal digital assistants), and so on. USB (universal serial bus) flash drives or USB sticks have become the most portable medium for carrying data between personal computers; they have a flash memory storage device integrated with a USB interface.

Optical Drives. The most popular form of optical removable storage is CDs (compact disks) and DVDs. CDs have a 700-MB capacity whereas DVDs have capacities ranging from 4.5 to 15 GB. CD-ROM(compact disk – read only memory) disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten. The version of compact and digital video disks called CD-R (compact disk recordable) and DVD-R or DVD+R, which are also known as WORM (write-once-read-many) disks, are a form of optical storage used for archiving data; they allow data to be written once and read any number of times without the possibility of erasing. They hold about half a gigabyte of data per disk and last much longer than magnetic disks.[3] A higher capacity format for DVDs called Blu-ray DVD can store 27 GB per layer, or 54 GB in a two-layer disk. **Optical jukebox memories** use an array of CD-ROM platters, which are loaded onto drives on demand. Although optical jukeboxes have capacities in the hundreds of gigabytes, their retrieval times are in the hundreds of milliseconds, quite a bit slower than magnetic disks. This type of **tertiary storage** is continuing to decline because of the rapid decrease in cost and the increase in capacities of magnetic disks. Most personal computer disk drives now read CD-ROM and DVD disks. Typically, drives are CD-R (compact disk recordable) that can create CD-ROMs and audio CDs, as well as record on DVDs.

Magnetic Tapes. Finally, **magnetic tapes** are used for archiving and backup storage of data. **Tape jukeboxes**—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as **tertiary storage** to hold terabytes of data. For example, NASA's EOS (Earth Observation Satellite) system stores archived databases in this fashion.

Many large organizations are using terabyte-sized databases. The term **very large database** can no longer be precisely defined because disk storage capacities are on

---

[3]Their rotational speeds are lower (around 400 rpm), giving higher latency delays and low transfer rates (around 100 to 200 KB/second) for a 1X drive. nX drives (e.g., 16X (n = 16)) are supposed to give n times higher transfer rate by multiplying the rpm n times. The 1X DVD transfer rate is about 1.385 MB/s.

Table 16.1   Types of Storage with Capacity, Access Time, Max Bandwidth (Transfer Speed), and Commodity Cost

Type	Capacity*	Access Time	Max Bandwidth	Commodity Prices (2014)**
Main Memory- RAM	4GB–1TB	30ns	35GB/sec	$100–$20K
Flash Memory- SSD	64 GB–1TB	50μs	750MB/sec	$50–$600
Flash Memory- USB stick	4GB–512GB	100μs	50MB/sec	$2–$200
Magnetic Disk	400 GB–8TB	10ms	200MB/sec	$70–$500
Optical Storage	50GB–100GB	180ms	72MB/sec	$100
Magnetic Tape	2.5TB–8.5TB	10s–80s	40–250MB/sec	$2.5K–$30K
Tape jukebox	25TB–2,100,000TB	10s–80s	250MB/sec–1.2PB/sec	$3K–$1M+

*Capacities are based on commercially available popular units in 2014.

**Costs are based on commodity online marketplaces.

the rise and costs are declining. Soon the term *very large database* may be reserved for databases containing hundreds of terabytes or petabytes.

To summarize, a hierarchy of storage devices and storage systems is available today for storage of data. Depending upon the intended use and application requirements, data is kept in one or more levels of this hierarchy. Table 16.1 summarizes the current state of these devices and systems and shows the range of capacities, average access times, bandwidths (transfer speeds), and costs on the open commodity market. Cost of storage is generally going down at all levels of this hierarchy.

16.1.2 Storage Organization of Databases

Databases typically store large amounts of data that must persist over long periods of time, and hence the data is often referred to as **persistent data**. Parts of this data are accessed and processed repeatedly during the storage period. This contrasts with the notion of **transient data**, which persists for only a limited time during program execution. Most databases are stored permanently (or *persistently*) on magnetic disk secondary storage, for the following reasons:

- Generally, databases are too large to fit entirely in main memory.[4]

- The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as **nonvolatile storage**, whereas main memory is often called **volatile storage**.

- The cost of storage per unit of data is an order of magnitude less for disk secondary storage than for primary storage.

---

[4]This statement is being challenged by recent developments in main memory database systems. Examles of prominent commercial systems include HANA by SAP and TIMESTEN by Oracle.

Some of the newer technologies—such as solid-state drive (SSD) disks are likely to provide viable alternatives to the use of magnetic disks. In the future, databases may therefore reside at different levels of the memory hierarchy from those described in Section 16.1.1. The levels may range from the highest speed main memory level storage to the tape jukebox low speed offline storage. However, it is anticipated that magnetic disks will continue to be the primary medium of choice for large databases for years to come. Hence, it is important to study and understand the properties and characteristics of magnetic disks and the way data files can be organized on disk in order to design effective databases with acceptable performance.

Magnetic tapes are frequently used as a storage medium for backing up databases because storage on tape costs much less than storage on disk. With some intervention by an operator—or an automatic loading device—tapes or optical removable disks must be loaded and read before the data becomes available for processing. In contrast, disks are **online** devices that can be accessed directly at any time.

The techniques used to store large amounts of structured data on disk are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific DBMS. Usually, the DBMS has several options available for organizing the data. The process of **physical database design** involves choosing the particular data organization techniques that best suit the given application requirements from among the options. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as **files** of **records**. Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently when they are needed. We will discuss some of the techniques for making disk access more efficient in Section 17.2.2.

There are several **primary file organizations**, which determine how the file records are *physically placed* on the disk, *and hence how the records can be accessed.* A *heap file* (or *unordered file*) places the records on disk in no particular order by appending new records at the end of the file, whereas a *sorted file* (or *sequential file*) keeps the records ordered by the value of a particular field (called the *sort key*). A *hashed file* uses a hash function applied to a particular field (called the *hash key*) to determine a record's placement on disk. Other primary file organizations, such as *B-trees,* use tree structures. We discuss primary file organizations in Sections 16.6 through 16.9. A **secondary organization** or **auxiliary access structure** allows efficient access to file records based on *alternate fields* than those that have been used for the primary file organization. Most of these exist as indexes and will be discussed in Chapter 17.

16.2 Secondary Storage Devices

In this section, we describe some characteristics of magnetic disk and magnetic tape storage devices. Readers who have already studied these devices may simply browse through this section.

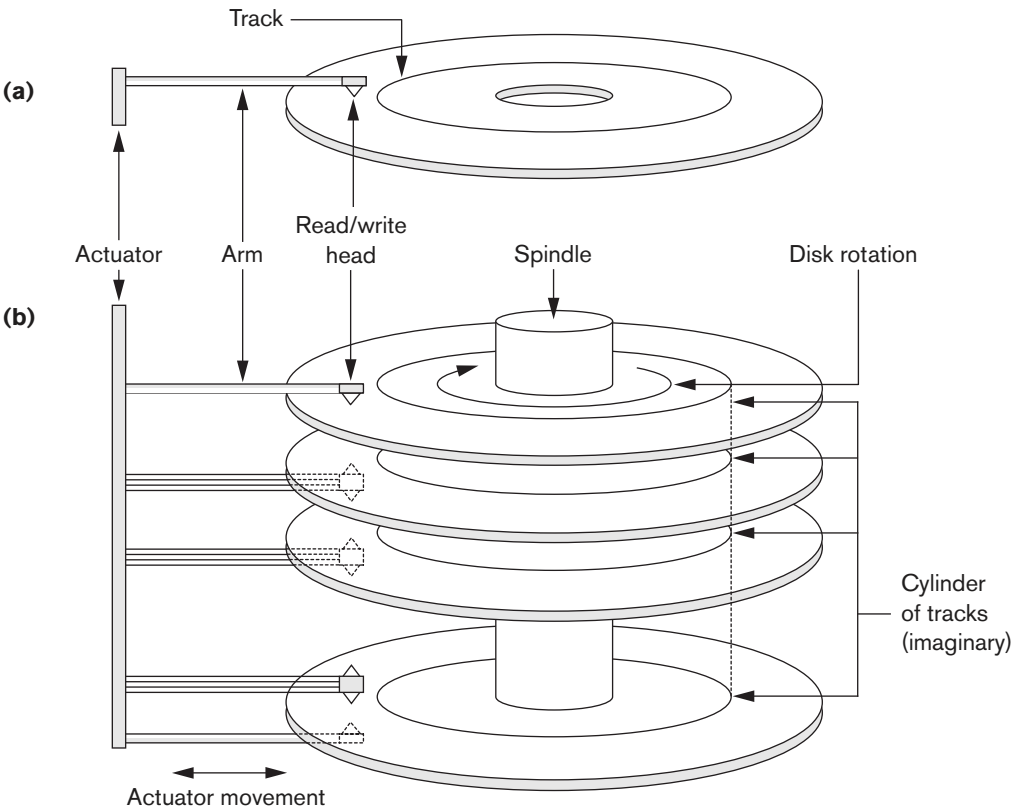### 16.2.1 Hardware Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The device that holds the disks is referred to as a **hard disk drive**, or **HDD**. The most basic unit of data on the disk is a single **bit** of information. By magnetizing an area on a disk in certain ways, one can make that area represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into **bytes** (or **characters**). Byte sizes are typically 4 to 8 bits, depending on the computer and the device; 8 bits is the most common. We assume that one character is stored in a single byte, and we use the terms *byte* and *character* interchangeably. The **capacity** of a disk is the number of bytes it can store, which is usually very large. Small floppy disks were used with laptops and desktops for many years—they contained a single disk typically holding from 400 KB to 1.5 MB; they are almost completely out of circulation. Hard disks for personal computers currently hold from several hundred gigabytes up to a few terabytes; and large disk packs used with servers and mainframes have capacities of hundreds of gigabytes. Disk capacities continue to grow as technology improves.

Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure 16.1(a), and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on one of its surfaces only and **double-sided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack**, as shown in Figure 16.1(b), which may include many disks and therefore many surfaces. The two most common form factors are 3.5 and 2.5 inch diameter. Information is stored on a disk surface in concentric circles of *small width*,[5] each having a distinct diameter. Each circle is called a **track**. In disk packs, tracks with the same diameter on the various surfaces are called a **cylinder** because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on one cylinder can be retrieved much faster than if it were distributed among different cylinders.

The number of tracks on a disk ranges from a few thousand to 152,000 on the disk drives shown in Table 16.2, and the capacity of each track typically ranges from tens of kilobytes to 150 Kbytes. Because a track usually contains a large amount of information, it is divided into smaller blocks or sectors. The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed. One type of sector organization, as shown in Figure 16.2(a), calls a portion of a track that subtends a fixed angle at the center a sector. Several other sector organizations are possible, one of which is to have the sectors subtend smaller angles at the center as one moves

---

[5]In some disks, the circles are now connected into a kind of continuous spiral.

Figure 16.1
(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.

Figure 16.2
Different sector
organizations on disk.
(a) Sectors subtending
a fixed angle.
(b) Sectors maintaining
a uniform recording
density.

away, thus maintaining a uniform density of recording, as shown in Figure 16.2(b). A technique called ZBR (zone bit recording) allows a range of cylinders to have the same number of sectors per arc. For example, cylinders 0–99 may have one sector per track, 100–199 may have two per track, and so on. A common sector size is 512 bytes. Not all disks have their tracks divided into sectors.

The division of a track into equal-sized **disk blocks** (or **pages**) is set by the operating system during disk **formatting** (or **initialization**). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range

Table 16.2   Specifications of Typical High-End Enterprise Disks from Seagate (a) Seagate Enterprise Performance 10 K HDD - 1200 GB

Specifications	1200GB
SED Model Number	ST1200MM0017
SED FIPS 140-2 Model Number	ST1200MM0027
Model Name	Enterprise Performance 10K HDD v7
Interface	6Gb/s SAS
Capacity	
Formatted 512 Bytes/Sector (GB)	1200
External Transfer Rate (MB/s)	600
Performance	
Spindle Speed (RPM)	10K
Average Latency (ms)	2.9
Sustained Transfer Rate Outer to Inner Diameter (MB/s)	204 to 125
Cache, Multisegmented (MB)	64
Configuration/Reliability	
Disks	4
Heads	8
Nonrecoverable Read Errors per Bits Read	1 per 10E16
Annualized Failure Rate (AFR)	0.44%
Physical	
Height (in/mm, max)	0.591/15.00
Width (in/mm, max)	2.760/70.10
Depth (in/mm, max)	3.955/100.45
Weight (lb/kg)	0.450/0.204

Courtesy Seagate Technology

(*Continued*)

Table 16.2 (b) Internal Drive Characteristics of 300 GB–900 GB Seagate Drives

	ST900MM0006 ST900MM0026 ST900MM0046 ST900MM0036	ST600MM0006 ST600MM0026 ST600MM0046	ST450MM0006 ST450MM0026 ST450MM0046	ST300MM0006 ST300MM0026 ST300MM0046	
Drive capacity	900	600	450	300	GB (formatted, rounded off value)
Read/write data heads	6	4	3	2	
Bytes per track	997.9	997.9	997.9	997.9	KBytes (avg, rounded off values)
Bytes per surface	151,674	151,674	151,674	151,674	MB (unformatted, rounded off value)
Tracks per surface (total)	152	152	152	152	KTracks (user accessible)
Tracks per inch	279	279	279	279	KTPI (average)
Peak bits per inch	1925	1925	1925	1925	KBPI
Areal density	538	538	538	538	Gb/in2
Disk rotation speed	10K	10K	10K	10K	rpm
Avg rotational latency	2.9	2.9	2.9	2.9	ms

from 512 to 8192 bytes. A disk with hard-coded sectors often has the sectors subdivided or combined into blocks during initialization. Blocks are separated by fixed-size **interblock gaps**, which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each interblock gap. Table 16.2 illustrates the specifications of typical disks used on large servers in industry. The 10K prefix on disk names refers to the rotational speeds in rpm (revolutions per minute.

There is continuous improvement in the storage capacity and transfer rates associated with disks; they are also progressively getting cheaper—currently costing only a fraction of a dollar per megabyte of disk storage. Costs are going down so rapidly that costs as low as $100/TB are already on the market.

A disk is a *random access* addressable device. Transfer of data between main memory and disk takes place in units of disk blocks. The **hardware address** of a block— a combination of a cylinder number, track number (surface number within the cylinder on which the track is located), and block number (within the track)—is supplied to the disk I/O (input/output) hardware. In many modern disk drives, a single number called LBA (logical block address), which is a number between 0 and n (assuming the total capacity of the disk is $n + 1$ blocks), is mapped automatically to the right block by the disk drive controller. The address of a **buffer**—a contiguous

reserved area in main storage that holds one disk block—is also provided. For a **read** command, the disk block is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a **cluster**, may be transferred as a unit. In this case, the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head**, which is part of a system called a **disk drive**. A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm**. Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface, as shown in Figure 16.1(b). All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.

Disk drives for hard disks rotate the disk pack continuously at a constant speed (typically ranging between 5,400 and 15,000 rpm). Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called **fixed-head** disks, whereas disk units with an actuator are called **movable-head disks**. For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is high, so fixed-head disks are not commonly used.

Interfacing Disk Drives to Computer Systems. A **disk controller**, typically embedded in the disk drive, controls the disk drive and interfaces it to the computer system. One of the standard interfaces used for disk drives on PCs and workstations was called **SCSI** (small computer system interface). Today to connect HDDs, CDs, and DVDs to a computer, the interface of choice is SATA. **SATA** stands for serial ATA, wherein ATA represents attachment; so SATA becomes serial AT attachment. It has its origin in PC/AT attachment, which referred to the direct attachment to the 16-bit bus introduced by IBM. The AT referred to advanced technology but is not used in the expansion of SATA due to trademark issues. Another popular interface used today is called **SAS** (serial attached SCSI). SATA was introduced in 2002 and allows the disk controller to be in the disk drive; only a simple circuit is required on the motherboard. SATA transfer speeds underwent an evolution from 2002 to 2008, going from 1.5 Gbps (gigabits per second) to 6 Gbps. SATA is now called NL-SAS for nearline SAS. The largest 3.5-inch SATA and SAS drives are 8TB, whereas 2.5-inch SAS drives are smaller and go up to 1.2TB. The 3.5-inch drives use 7,200 or 10,000 rpm speed whereas 2.5-inch drives use up to 15,000 rpm. In terms of IOPs (input/output operations) per second as a price to performance index, SAS is considered superior to SATA.

The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to take place. To transfer a disk block, given its address, the disk controller must first mechanically position the

read/write head on the correct track. The time required to do this is called the **seek time**. Typical seek times are 5 to 10 msec on desktops and 3 to 8 msec on servers. Following that, there is another delay—called the **rotational delay** or **latency**—while the beginning of the desired block rotates into position under the read/write head. It depends on the rpm of the disk. For example, at 15,000 rpm, the time per rotation is 4 msec and the average rotational delay is the time per half revolution, or 2 msec. At 10,000 rpm the average rotational delay increases to 3 msec. Finally, some additional time is needed to transfer the data; this is called the **block transfer time**. Hence, the total time needed to locate and transfer an arbitrary block, given its address, is the sum of the seek time, rotational delay, and block transfer time. The seek time and rotational delay are usually much larger than the block transfer time. To make the transfer of multiple blocks more efficient, it is common to transfer several consecutive blocks on the same track or cylinder. This eliminates the seek time and rotational delay for all but the first block and can result in a substantial saving of time when numerous contiguous blocks are transferred. Usually, the disk manufacturer provides a **bulk transfer rate** for calculating the time required to transfer consecutive blocks. Appendix B contains a discussion of these and other disk parameters.

The time needed to locate and transfer a disk block is in the order of milliseconds, usually ranging from 9 to 60 msec. For contiguous blocks, locating the first block takes from 9 to 60 msec, but transferring subsequent blocks may take only 0.4 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on a disk. In any case, a transfer time in the order of milliseconds is considered high compared with the time required to process data in main memory by current CPUs. Hence, locating data on disk is a *major bottleneck* in database applications. The file structures we discuss here and in Chapter 17 attempt to *minimize the number of block transfers* needed to locate and transfer the required data from disk to main memory. Placing "related information" on contiguous blocks is the basic goal of any storage organization on disk.

16.2.2 Making Data Access More Efficient on Disk

In this subsection, we list some of the commonly used techniques to make accessing data more efficient on HDDs.

1. **Buffering of data:** In order to deal with the incompatibility of speeds between a CPU and the electromechanical device such as an HDD, which is inherently slower, buffering of data is done in memory so that new data can be held in a buffer while old data is processed by an application. We discuss the double buffering strategy followed by general issues of buffer management and buffer replacement strategies in Section 16.3.

2. **Proper organization of data on disk:** Given the structure and organization of data on disk, it is advantageous to keep related data on contiguous blocks; when multiple cylinders are needed by a relation, contiguous cylinders should be used. Doing so avoids unnecessary movement of the read/write arm and related seek times.

3. **Reading data ahead of request:** To minimize seek times, whenever a block is read into the buffer, blocks from the rest of the track can also be read even though they may not have been requested yet. This works well for applications that are likely to need consecutive blocks; for random block reads this strategy is counterproductive.

4. **Proper scheduling of I/O requests:** If it is necessary to read several blocks from disk, total access time can be minimized by scheduling them so that the arm moves only in one direction and picks up the blocks along its movement. One popular algorithm is called the elevator algorithm; this algorithm mimics the behavior of an elevator that schedules requests on multiple floors in a proper sequence. In this way, the arm can service requests along its outward and inward movements without much disruption.

5. **Use of log disks to temporarily hold writes:** A single disk may be assigned to just one function called logging of writes. All blocks to be written can go to that disk sequentially, thus eliminating any seek time. This works much faster than doing the writes to a file at random locations, which requires a seek for each write. The log disk can order these writes in (cylinder, track) ordering to minimize arm movement when writing. Actually, the log disk can only be an area (extent) of a disk. Having the data file and the log file on the same disk is a cheaper solution but compromises performance. Although the idea of a log disk can improve write performance, it is not feasible for most real-life application data.

6. **Use of SSDs or flash memory for recovery purposes:** In applications where updates occur with high frequency, updates can be lost from main memory if the system crashes. A preventive measure would be to increase the speed of updates/writes to disk. One possible approach involves writing the updates to a nonvolatile SSD buffer, which may be a flash memory or battery-operated DRAM, both of which operate at must faster speeds (see Table 16.1). The disk controller then updates the data file during its idle time and also when the buffer becomes full. During recovery from a crash, unwritten SSD buffers must be written to the data file on HDD. For further discussion of recovery and logs, consult Chapter 22.

16.2.3 SolidState Device (SSD) Storage

This type of storage is sometimes known as flash storage because it is based on the flash memory technology, which we discussed in Section 16.1.1.

The recent trend is to use flash memories as an intermediate layer between main memory and secondary rotating storage in the form of magnetic disks (HDDs). Since they resemble disks in terms of the ability to store data in secondary storage without the need for continuous power supply, they are called **solid-state disks** or **solid-state drives (SSDs)**. We will discuss SSDs in general terms first and then comment on their use at the enterprise level, where they are sometimes referred to as **enterprise flash drives (EFDs),** a term first introduced by EMC Corporation.

The main component of an SSD is a controller and a set of interconnected flash memory cards. Use of NAND flash memory is most common. Using form factors compatible with 3.5 inch or 2.5 inch HDDs makes SSDs pluggable into slots already available for mounting HDDs on laptops and servers. For ultrabooks, tablets, and the like, card-based form factors such as mSATA and M.2 are being standardized. Interfaces like **SATA express** have been created to keep up with advancements in SSDs. Because there are no moving parts, the unit is more rugged, runs silently, is faster in terms of access time and provides higher transfer rates than HDD. As opposed to HDDs, where related data from the same relation must be placed on contiguous blocks, preferably on contiguous cylinders, there is no restriction on placement of data on an SSD since any address is directly addressable. As a result, the data is less likely to be fragmented; hence no reorganization is needed. Typically, when a write to disk occurs on an HDD, the same block is overwritten with new data. In SDDs, the data is written to different NAND cells to attain **wear-leveling**, which prolongs the life of the SSD. The main issue preventing a wide-scale adoption of SSDs today is their prohibitive cost (see Table 16.1), which tends to be about 70 to 80 cents per GB as opposed to about 15 to 20 cents per GB for HDDs.

In addition to flash memory, DRAM-based SSDs are also available. They are costlier than flash memory, but they offer faster access times of around 10 μs (microseconds) as opposed to 100 μs for flash. Their main drawback is that they need an internal battery or an adapter to supply power.

As an example of an enterprise level SSD, we can consider CISCO's UCS (Unified Computing System©) Invicta series SSDs. They have made it possible to deploy SSDs at the data center level to unify workloads of all types, including databases and virtual desktop infrastructure (VDI), and to enable a cost-effective, energy-efficient, and space-saving solution. CISCO's claim is that Invicta SSDs offer a better price-to-performance ratio to applications in a multitenant, multinetworked architecture because of the advantages of SSDs stated above. CISCO states that typically four times as many HDD drives may be needed to match an SSD-based RAID in performance.[6] The SSD configuration can have a capacity from 6 to 144 TB, with up to 1.2 million I/O operations/second, and a bandwidth of up to 7.2 GB/sec with an average latency of 200 μs.[7] Modern data centers are undergoing rapid transformation and must provide real-time response using cloud-based architectures. In this environment, SSDs are likely to play a major role.

16.2.4 Magnetic Tape Storage Devices

Disks are **random access** secondary storage devices because an arbitrary disk block may be accessed *at random* once we specify its address. Magnetic tapes are sequential access devices; to access the *n*th block on tape, first we must scan the preceding

---

[6]Based on the CISCO White Paper (CISCO, 2014)

[7]Data sheet for CISCO UCS Invicta Scaling System.

$n - 1$ blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audiotapes or videotapes. A tape drive is required to read the data from or write the data to a **tape reel**. Usually, each group of bits that forms a byte is stored across the tape, and the bytes themselves are stored consecutively on the tape.

A read/write head is used to read or write data on tape. Data records on tape are also stored in blocks—although the blocks may be substantially larger than those for disks, and interblock gaps are also quite large. With typical tape densities of 1,600 to 6,250 bytes per inch, a typical interblock gap[8] of 0.6 inch corresponds to 960 to 3,750 bytes of wasted storage space. It is customary to group many records together in one block for better space utilization.

The main characteristic of a tape is its requirement that we access the data blocks in **sequential order**. To get to a block in the middle of a reel of tape, the tape is mounted and then scanned until the required block gets under the read/write head. For this reason, tape access can be slow and tapes are not used to store online data, except for some specialized applications. However, tapes serve a very important function— **backing up** the database. One reason for backup is to keep copies of disk files in case the data is lost due to a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape. For many online critical applications, such as airline reservation systems, to avoid any downtime, mirrored systems are used to keep three sets of identical disks—two in online operation and one as backup. Here, offline disks become a backup device. The three are rotated so that they can be switched in case there is a failure on one of the live disk drives. Tapes can also be used to store excessively large database files. Database files that are seldom used or are outdated but required for historical recordkeeping can be **archived** on tape. Originally, half-inch reel tape drives were used for data storage employing the so-called nine-track tapes. Later, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 GB, as well as 4-mm helical scan data cartridges and writable CDs and DVDs, became popular media for backing up data files from PCs and workstations. They are also used for storing images and system libraries.

Backing up enterprise databases so that no transaction information is lost is a major undertaking. Tape libraries were in vogue and featured slots for several hundred cartridges; these tape libraries used digital and superdigital linear tapes (DLTs and SDLTs), both of which have capacities in the hundreds of gigabytes and record data on linear tracks. These tape libraries are no longer in further development. The LTO (Linear Tape Open) consortium set up by IBM, HP, and Seagate released the latest LTO-6 standard in 2012 for tapes. It uses $\frac{1}{2}$-inch-wide magnetic tapes like those used in earlier tape drives but in a somewhat smaller, single-reel enclosed cartridge. Current generation of libraries use LTO-6 drives, at 2.5-TB cartridge with 160 MB/s transfer rate. Average seek time is about 80 seconds. The T10000D drive of Oracle/StorageTek handles 8.5 TB on a single cartridge with transfer rate upto 252 MB/s.

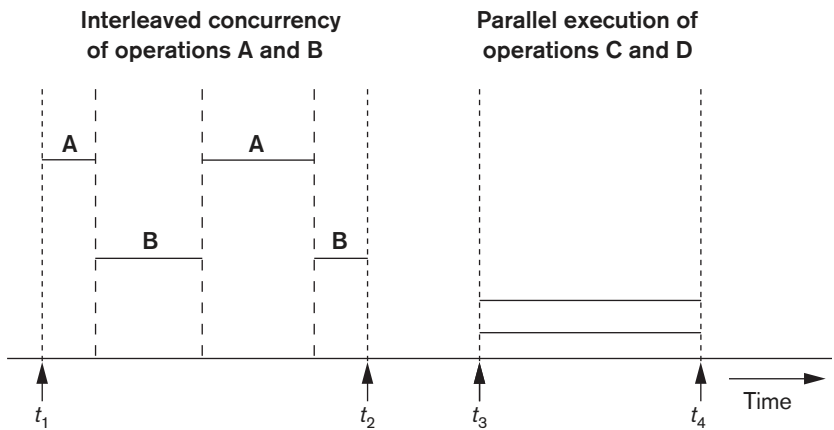[8]Called *interrecord gaps* in tape terminology.

Robotic arms write on multiple cartridges in parallel using multiple tape drives and automatic labeling software to identify the backup cartridges. An example of a giant library is the SL8500 model of Sun Storage Technology. The SL8500 scales from 1,450 to just over 10,000 slots and from 1 to 64 tape drives within each library. It accepts both DLT/SDLT and LTO tapes. Up to 10 SL8500s can be connected within a single library complex for over 100,000 slots and up to 640 drives. With 100,000 slots, the SL8500 can store 2.1 exabytes (exabyte = 1,000 petabytes, or million TB = $10^{**}18$ bytes). We defer the discussion of disk storage technology called RAID, and of storage area networks, network-attached storage, and iSCSI storage systems, to the end of the chapter.

16.3 Buffering of Blocks

When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.
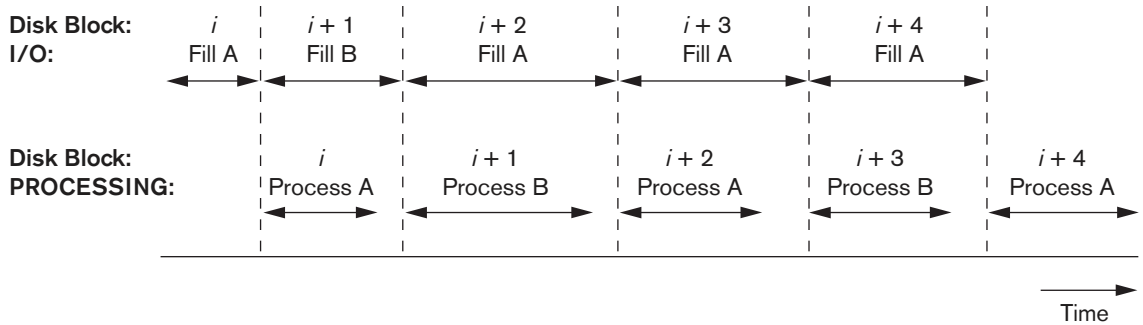
Figure 16.3 illustrates how two processes can proceed in parallel. Processes A and B are running **concurrently** in an **interleaved** fashion, whereas processes C and D are running **concurrently** in a **parallel** fashion. When a single CPU controls multiple processes, parallel execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when processes can run concurrently in a parallel fashion, either because a separate disk I/O processor is available or because multiple CPU processors exist.

Figure 16.4 illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to

Figure 16.3
Interleaved concurrency versus parallel execution.

Disk Block: I/O:	i Fill A	$i+1$ Fill B	$i+2$ Fill A	$i+3$ Fill A	$i+4$ Fill A

Disk Block: PROCESSING:	i Process A	$i+1$ Process B	$i+2$ Process A	$i+3$ Process B	$i+4$ Process A

Time

Figure 16.4
Use of two buffers, A and B, for reading from disk.

read the next block and fill a buffer. The CPU can start processing a block once its transfer to main memory is completed; at the same time, the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called **double buffering** and can also be used to read a continuous stream of blocks from disk to memory. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

16.3.1 Buffer Management

Buffer management and Replacement Strategies. For most large database files containing millions of pages, it is not possible to bring all of the data into main memory at the same time. We alluded to double buffering as a technique whereby we can gain efficiency in terms of performing the I/O operation between the disk and main memory into one buffer area concurrently with processing the data from another buffer. The actual management of buffers and decisions about what buffers to use to place a newly read page in the buffer is a more complex process. We use the term **buffer** to refer to a part of main memory that is available to receive blocks or pages of data from disk.[9] **Buffer manager** is a software component of a DBMS that responds to requests for data and decides what buffer to use and what pages to replace in the buffer to accommodate the newly requested blocks. The buffer manager views the available main memory storage as a **buffer pool**, which has a collection of pages. The size of the shared buffer pool is typically a parameter for the DBMS controlled by DBAs. In this section, we briefly discuss the workings of the buffer manager and discuss a few replacement strategies.

---

[9]We use the terms page and block interchangeably in the current context.

There are two kinds of buffer managers; the first kind controls the main memory directly, as in most RDBMSs. The second kind allocates buffers in virtual memory, which allows the control to transfer to the operating system (OS). The OS in turn controls which buffers are actually in main memory and which ones are on disk under the control of OS. This second kind of buffer manager is common in main memory database systems and some object-oriented DBMSs. The overall goal of the buffer manager is twofold: (1) to maximize the probability that the requested page is found in main memory, and (2) in case of reading a new disk block from disk, to find a page to replace that will cause the least harm in the sense that it will not be required shortly again.

To enable its operation, the buffer manager keeps two types of information on hand about each page in the buffer pool:

1. A **pin-count**: the number of times that page has been requested, or the number of current users of that page. If this count falls to zero, the page is considered **unpinned**. Initially the pin-count for every page is set to zero. Incrementing the pin-count is called **pinning**. In general, a pinned block should not be allowed to be written to disk.

2. A **dirty bit**, which is initially set to zero for all pages but is set to 1 whenever that page is updated by any application program.

In terms of storage management, the buffer manager has the following responsibility: It must make sure that the number of buffers fits in main memory. If the requested amount of data exceeds available buffer space, the buffer manager must select what buffers must be emptied, as governed by the buffer replacement policy in force. If the buffer manager allocates space in virtual memory and all buffers in use exceed the actual main memory, then the common operating system problem of "thrashing" happens and pages get moved back and forth into the swap space on disk without performing useful work.

When a certain page is requested, the buffer manager takes following actions: it checks if the requested page is already in a buffer in the buffer pool; if so, it increments its pin-count and releases the page. If the page is not in the buffer pool, the buffer manager does the following:

a. It chooses a page for replacement, using the replacement policy, and increments its pin-count.

b. If the dirty bit of the replacement page is on, the buffer manager writes that page to disk by replacing its old copy on disk. If the dirty bit is not on, this page is not modified and the buffer manager is not required to write it back to disk.

c. It reads the requested page into the space just freed up.

d. The main memory address of the new page is passed to the requesting application.

If there is no unpinned page available in the buffer pool and the requested page is not available in the buffer pool, the buffer manager may have to wait until a page gets released. A transaction requesting this page may go into a wait state or may even be aborted.

16.3.2 Buffer Replacement Strategies:

The following are some popular replacement strategies that are similar to those used elsewhere, such as in operating systems:

1. Least recently used (LRU): The strategy here is to throw out that page that has not been used (read or written) for the longest time. This requires the buffer manager to maintain a table where it records the time every time a page in a buffer is accessed. Whereas this constitutes an overhead, the strategy works well because for a buffer that is not used for a long time, its chance of being accessed again is small.

2. Clock policy: This is a round-robin variant of the LRU policy. Imagine the buffers are arranged like a circle similar to a clock. Each buffer has a flag with a 0 or 1 value. Buffers with a 0 are vulnerable and may be used for replacement and their contents read back to disk. Buffers with a 1 are not vulnerable. When a block is read into a buffer, the flag is set to 1. When the buffer is accessed, the flag is set to 1 also. The clock hand is positioned on a "current buffer." When the buffer manager needs a buffer for a new block, it rotates the hand until it finds a buffer with a 0 and uses that to read and place the new block. (If the dirty bit is on for the page being replaced, that page will be written to disk, thus overwriting the old page at its address on disk.) If the clock hand passes buffers with 1s, it sets them to a zero. Thus, a block is replaced from its buffer only if it is not accessed until the hand completes a rotation and returns to it and finds the block with the 0 that it set the last time.

3. First-in-first-out (FIFO): Under this policy, when a buffer is required, the one that has been occupied the longest by a page is used for replacement. Under this policy, the manager notes the time each page gets loaded into a buffer; but it does not have to keep track of the time pages are accessed. Although FIFO needs less maintenance than LRU, it can work counter to desirable behavior. A block that remains in the buffer for a long time because it is needed continuously, such as a root block of an index, may be thrown out but may be immediately required to be brought back.

LRU and clock policies are not the best policies for database applications if they require sequential scans of data and the file cannot fit into the buffer at one time. There are also situations when certain pages in buffers cannot be thrown out and written out to disk because certain other pinned pages point to those pages. Also, policies like FIFO can be modified to make sure that pinned blocks, such as root block of an index, are allowed to remain in the buffer. Modification of the clock policy also exists where important buffers can be set to higher values than 1 and therefore will not be subjected to replacement for several rotations of the hand. There are also situations when the DBMS has the ability to write certain blocks to disk even when the space occupied by those blocks is not needed. This is called **force-writing** and occurs typically when log records have to be written to disk ahead of the modified pages in a transaction for recovery purposes. (See Chapter 22.) There are some other replacement strategies such as **MRU (most recently used)**

that work well for certain types of database transactions, such as when a block that is used most recently is not needed until all the remaining blocks in the relation are processed.

16.4 Placing File Records on Disk

Data in a database is regarded as a set of records organized into a set of files. In this section, we define the concepts of records, record types, and files. Then we discuss techniques for placing file records on disk. Note that henceforth in this chapter we will be referring to the random access persistent secondary storage as "disk drive" or "disk." The disk may be in different forms; for example, magnetic disks with rotational memory or solid-state disks with electronic access and no mechanical delays.

16.4.1 Records and Record Types

Data is usually stored in the form of **records**. Each record consists of a collection of related data **values** or **items**, where each value is formed of one or more bytes and corresponds to a particular **field** of the record. Records usually describe entities and their attributes. For example, an EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as Name, Birth_date, Salary, or Supervisor. A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition. A **data type**, associated with each field, specifies the types of values a field can take.

The data type of a field is usually one of the standard data types used in programming. These include numeric (integer, long integer, or floating point), string of characters (fixed-length or varying), Boolean (having 0 and 1 or TRUE and FALSE values only), and sometimes specially coded **date** and **time** data types. The number of bytes required for each data type is fixed for a given computer system. An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 10 bytes (assuming a format of YYYY-MM-DD), and a fixed-length string of k characters k bytes. Variable-length strings may require as many bytes as there are characters in each field value. For example, an EMPLOYEE record type may be defined—using the C programming language notation—as the following structure:

```
struct employee{
    char name[30];
    char ssn[9];
    int salary;
    int job_code;
    char department[20];
} ;
```

In some database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as **BLOB**s (binary large objects). A BLOB data item is typically stored separately from its record in a pool of disk blocks, and

a pointer to the BLOB is included in the record. For storing free text, some DBMSs (e.g., Oracle, DB2, etc.) provide a data type called CLOB (character large object); some DBMSs call this data type text.
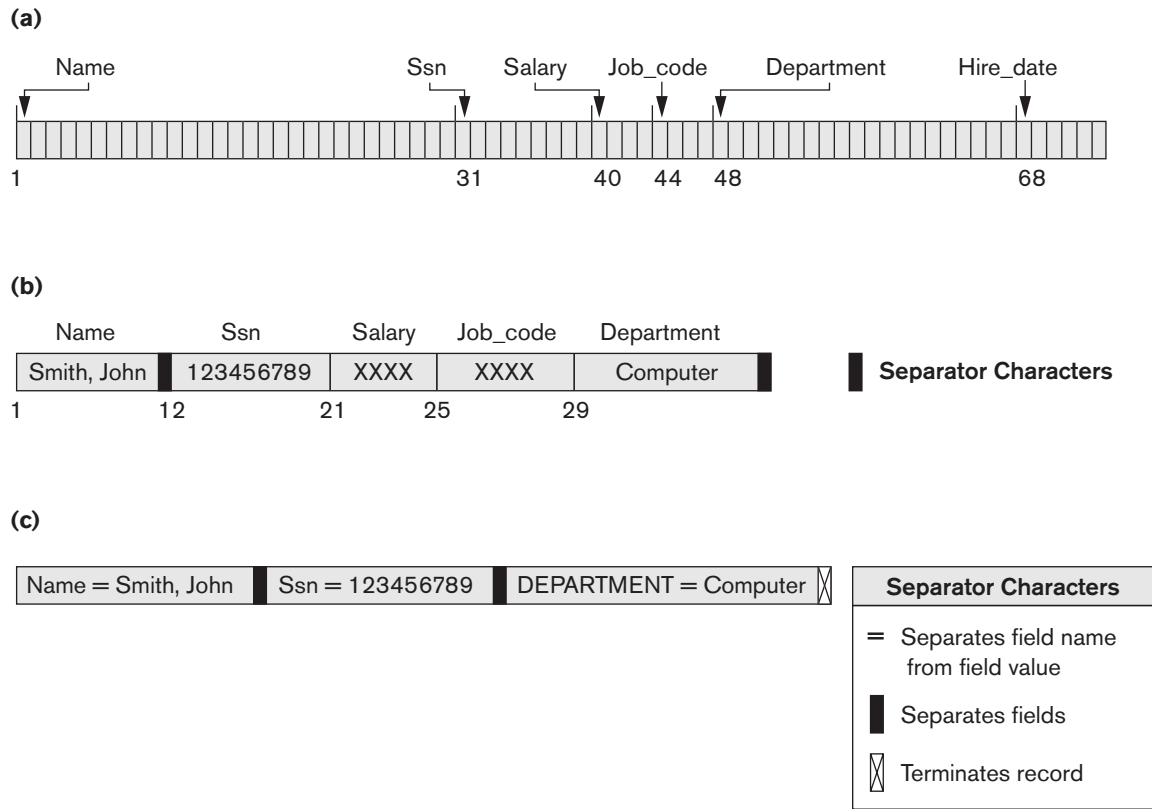
16.4.2 Files, Fixed-Length Records, and Variable-Length Records

A **file** is a *sequence* of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records**. If different records in the file have different sizes, the file is said to be made up of **variable-length records**. A file may have variable-length records for several reasons:

- The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the Name field of EMPLOYEE can be a variable-length field.

- The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a **repeating group**.

- The file records are of the same record type, but one or more of the fields are **optional**; that is, they may have values for some but not all of the file records (**optional fields**).

- The file contains records of *different record types* and hence of varying size (**mixed file**). This would occur if related records of different types were *clustered* (placed together) on disk blocks; for example, the GRADE_REPORT records of a particular student may be placed following that STUDENT's record.

The fixed-length EMPLOYEE records in Figure 16.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files. Notice that it is possible to represent a file that logically should have variable-length records as a fixed-length records file. For example, in the case of optional fields, we could have *every field* included in *every file record* but store a special NULL value if no value exists for that field. For a repeating field, we could allocate as many spaces in each record as the *maximum possible number of occurrences* of the field. In either case, space is wasted when certain records do not have values for all the physical spaces provided in each record. Now we consider other options for formatting records of a file of variable-length records.

For *variable-length fields,* each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special **separator** characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields, as shown in Figure 16.5(b), or we can store the length in bytes of the field in the record, preceding the field value.

(a)

(b)

(c)

Figure 16.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

A file of records with *optional fields* can be formatted in different ways. If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values. Three types of separator characters are used in Figure 16.5(c), although we could use the same separator character for the first two purposes—separating the field name from the field value and separating one field from the next field. A more practical option is to assign a short **field type** code—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs.

A *repeating field* needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field. Finally, for a file that includes *records of different types,* each record is preceded by a **record**

type indicator. Understandably, programs that process files of variable-length records—which are usually part of the file system and hence hidden from the typical programmers—need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.[10]
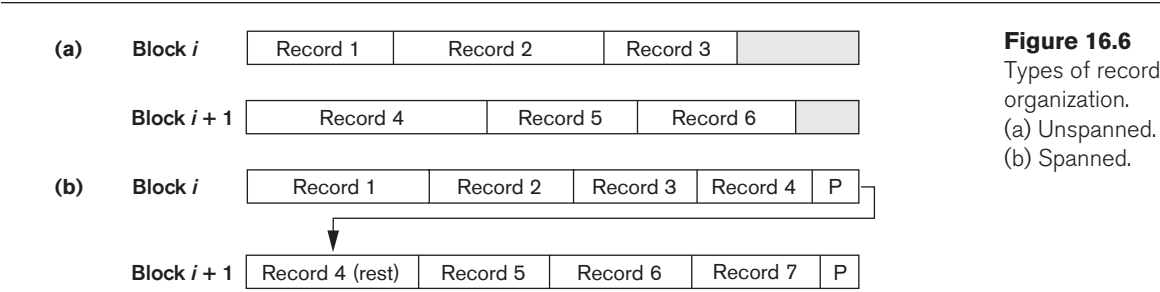
16.4.3 Record Blocking and Spanned versus Unspanned Records

The records of a file must be allocated to disk blocks because a block is the *unit of data transfer* between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block. Suppose that the block size is *B* bytes. For a file of fixed-length records of size *R* bytes, with $B \geq R$, we can fit $bfr = \lfloor B/R \rfloor$ records per block, where the $\lfloor (x) \rfloor$ (*floor function*) *rounds down* the number *x* to an integer. The value *bfr* is called the **blocking factor** for the file. In general, *R* may not divide *B* exactly, so we have some unused space in each block equal to

$$B - (bfr * R) \text{ bytes}$$

To utilize this unused space, we can store part of a record on one block and the rest on another. A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called **spanned** because records can span more than one block. Whenever a record is larger than a block, we *must* use a spanned organization. If records are not allowed to cross block boundaries, the organization is called **unspanned**. This is used with fixed-length records having $B > R$ because it makes each record start at a known location in the block, simplifying record processing. For variable-length records, either a spanned or an unspanned organization can be used. If the average record is large, it is advantageous to use spanning to reduce the lost space in each block. Figure 16.6 illustrates spanned versus unspanned organization.

For variable-length records using spanned organization, each block may store a different number of records. In this case, the blocking factor *bfr* represents the *average*

Figure 16.6
Types of record organization.
(a) Unspanned.
(b) Spanned.

---

[10]Other schemes are also possible for representing variable-length records.

number of records per block for the file. We can use *bfr* to calculate the number of blocks *b* needed for a file of *r* records:

$$b = \lceil (r/bfr) \rceil \text{ blocks}$$

where the $\lceil (x) \rceil$ (*ceiling function*) rounds the value *x* up to the next integer.

16.4.4 Allocating File Blocks on Disk

There are several standard techniques for allocating the blocks of a file on disk. In **contiguous allocation**, the file blocks are allocated to consecutive disk blocks. This makes reading the whole file very fast using double buffering, but it makes expanding the file difficult. In **linked allocation**, each file block contains a pointer to the next file block. This makes it easy to expand the file but makes it slow to read the whole file. A combination of the two allocates **clusters** of consecutive disk blocks, and the clusters are linked. Clusters are sometimes called **file segments** or **extents**. Another possibility is to use **indexed allocation**, where one or more **index blocks** contain pointers to the actual file blocks. It is also common to use combinations of these techniques.

16.4.5 File Headers

A **file header** or **file descriptor** contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, using the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a **linear search** through the file blocks. Each file block is copied into a buffer and searched until the record is located or all the file blocks have been searched unsuccessfully. This can be very time-consuming for a large file. The goal of a good file organization is to avoid linear search or full scan of the file and to locate the block that contains a desired record with a minimal number of block transfers.

16.5 Operations on Files

Operations on files are usually grouped into **retrieval operations** and **update operations**. The former do not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to **select** one or more records for retrieval, deletion, or modification based on a **selection condition** (or **filtering condition**), which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields Name, Ssn, Salary, Job_code, and Department. A **simple selection condition** may involve an equality comparison on some field value—for example, (Ssn = '123456789') or (Department = 'Research'). More complex conditions can involve other types of comparison operators, such as > or ≥ ; an example is (Salary ≥ 30000). The general case is to have an arbitrary Boolean expression on the fields of the file as the selection condition.

Search operations on files are generally based on simple selection conditions. A complex condition must be decomposed by the DBMS (or the programmer) to extract a simple condition that can be used to locate the records on disk. Each located record is then checked to determine whether it satisfies the full selection condition. For example, we may extract the simple condition (Department = 'Research') from the complex condition ((Salary ≥ 30000) AND (Department = 'Research')); each record satisfying (Department = 'Research') is located and then tested to see if it also satisfies (Salary ≥ 30000).

When several file records satisfy a search condition, the *first* record—with respect to the physical sequence of file records—is initially located and designated the **current record**. Subsequent search operations commence from this record and locate the *next* record in the file that satisfies the condition.

Actual operations for locating and accessing file records vary from system to system. In the following list, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access records by using these commands, so we sometimes refer to **program variables** in the following descriptions:

- **Open.** Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.

- **Reset.** Sets the file pointer of an open file to the beginning of the file.

- **Find (or Locate).** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the *current record.* Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.

- **Read (or Get).** Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.

- **FindNext.** Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record. Various forms of FindNext (for example, FindNext record within a current parent record, FindNext record of a given type, or FindNext record where a complex condition is met) are available in legacy DBMSs based on the hierarchical and network models.

■ **Delete.** Deletes the current record and (eventually) updates the file on disk to reflect the deletion.

■ **Modify.** Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.

■ **Insert.** Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.

■ **Close.** Completes the file access by releasing the buffers and performing any other needed cleanup operations.

The preceding (except for Open and Close) are called **record-at-a-time** operations because each operation applies to a single record. It is possible to streamline the operations Find, FindNext, and Read into a single operation, Scan, whose description is as follows:

■ **Scan.** If the file has just been opened or reset, *Scan* returns the first record; otherwise it returns the next record. If a condition is specified with the operation, the returned record is the first or next record satisfying the condition.

In database systems, additional **set-at-a-time** higher-level operations may be applied to a file. Examples of these are as follows:

■ **FindAll.** Locates *all* the records in the file that satisfy a search condition.

■ **Find (or Locate) *n*.** Searches for the first record that satisfies a search condition and then continues to locate the next $n - 1$ records satisfying the same condition. Transfers the blocks containing the *n* records to the main memory buffer (if not already there).

■ **FindOrdered.** Retrieves all the records in the file in some specified order.

■ **Reorganize.** Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

At this point, it is worthwhile to note the difference between the terms *file organization* and *access method*. A **file organization** refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An **access method**, on the other hand, provides a group of operations—such as those listed earlier—that can be applied to a file. In general, it is possible to apply several access methods to a file organized using a certain organization. Some access methods, though, can be applied only to files organized in certain ways. For example, we cannot apply an indexed access method to a file without an index (see Chapter 17).

Usually, we expect to use some search conditions more than others. Some files may be **static**, meaning that update operations are rarely performed; other, more **dynamic** files may change frequently, so update operations are constantly applied to them. If a file is not updatable by the end user, it is regarded as a read-only file.

Most data warehouses (see Chapter 29) predominantly contain read-only files. A successful file organization should perform as efficiently as possible the operations we expect to *apply frequently* to the file. For example, consider the EMPLOYEE file, as shown in Figure 16.5(a), which stores the records for current employees in a company. We expect to insert records (when employees are hired), delete records (when employees leave the company), and modify records (for example, when an employee's salary or job is changed). Deleting or modifying a record requires a selection condition to identify a particular record or set of records. Retrieving one or more records also requires a selection condition.

If users expect mainly to apply a search condition based on Ssn, the designer must choose a file organization that facilitates locating a record given its Ssn value. This may involve physically ordering the records by Ssn value or defining an index on Ssn (see Chapter 17). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks are grouped by department. For this application, it is best to order employee records by department and then by name within each department. The clustering of records into blocks and the organization of blocks on cylinders would now be different than before. However, this arrangement conflicts with ordering the records by Ssn values. If both applications are important, the designer should choose an organization that allows both operations to be done efficiently. Unfortunately, in many cases a single organization does not allow all needed operations on a file to be implemented efficiently. Since a file can be stored only once using one particular organization, the DBAs are often faced with making a difficult design choice about the file organization. They make it based on the expected importance and mix of retrieval and update operations.

In the following sections and in Chapter 17, we discuss methods for organizing records of a file on disk. Several general techniques, such as ordering, hashing, and indexing, are used to create access methods. Additionally, various general techniques for handling insertions and deletions work with many file organizations.

16.6 Files of Unordered Records (Heap Files)

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a **heap** or **pile file**.[11] This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 17. It is also used to collect and store data records for future use.

Inserting a new record is *very efficient.* The last disk block of the file is copied into a buffer, the new record is added, and the block is then **rewritten** back to disk. The address of the last file block is kept in the file header. However, searching for a record using any search condition involves a **linear search** through the file block by block—an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file

---

[11]Sometimes this organization is called a **sequential file**.

blocks before it finds the record. For a file of *b* blocks, this requires searching ($b/2$) blocks, on average. If no records or several records satisfy the search condition, the program must read and search all *b* blocks in the file.

To delete a record, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra byte or bit, called a **deletion marker**, stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value for the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic **reorganization** of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records. After such a reorganization, the blocks are filled to capacity once more. Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations.

We can use either spanned or unspanned organization for an unordered file, and it may be used with either fixed-length or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a modified record because the modified record may not fit in its old space on disk.

To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special techniques for **external sorting** are used (see Chapter 18).

For a file of unordered *fixed-length records* using *unspanned blocks* and *contiguous allocation,* it is straightforward to access any record by its **position** in the file. If the file records are numbered 0, 1, 2, … , $r - 1$ and the records in each block are numbered 0, 1, …, $bfr - 1$, where *bfr* is the blocking factor, then the *i*th record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the (i mod *bfr*)th record in that block. Such a file is often called a **relative** or **direct file** because records can easily be accessed directly by their relative positions. Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates the construction of access paths on the file, such as the indexes discussed in Chapter 17.

16.7 Files of Ordered Records (Sorted Files)

We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field**. This leads to an **ordered** or **sequential** file.[12] If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record—then the field is called the **ordering key** for the file. Figure 16.7

---

[12]The term *sequential file* has also been used to refer to unordered files, although it is more appropriate for ordered files.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
Block n−1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
Block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Figure 16.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with
Name as the ordering key field.

shows an ordered file with Name as the ordering key field (assuming that employees have distinct names).

Ordered records have some advantages over unordered files. First, reading the records in order of the ordering key values becomes extremely efficient because no sorting is required. The search condition may be of the type < key = value>, or a range condition such as < value1 < key < value2>. Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current one (unless the current record is the last one in the block). Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files. Ordered files are blocked and stored on contiguous cylinders to minimize the seek time.

A **binary search** for disk files can be done on the blocks rather than on the records. Suppose that the file has b blocks numbered 1, 2, …, b; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K. Assuming that disk addresses of the file blocks are available in the file header, the binary search can be described by Algorithm 16.1. A binary search usually accesses $\log_2(b)$ blocks, whether the record is found or not—an improvement over linear searches, where, on the average, $(b/2)$ blocks are accessed when the record is found and b blocks are accessed when the record is not found.

> **Algorithm 16.1.** Binary Search on an Ordering Key of a Disk File
> $l \leftarrow 1; u \leftarrow b;$ (*b is the number of file blocks*)
> while $(u \geq l)$ do
>   **begin** $i \leftarrow (l + u)$ div 2;
>   read block i of the file into the buffer;
>   if $K <$ (ordering key field value of the *first* record in block i)
>   then $u \leftarrow i - 1$
>   else if $K >$ (ordering key field value of the *last* record in block i)
>   then $l \leftarrow i + 1$
>   else if the record with ordering key field value $= K$ is in the buffer
>   then goto found
>   else goto notfound;
>   **end**;
>   goto notfound;

A search criterion involving the conditions >, <, ≥, and ≤ on the ordering field is efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 16.7, if the search criterion is (Name > 'G')—where > means *alphabetically before*—the records satisfying the search criterion are those from the beginning of the file up to the first record that has a Name value starting with the letter 'G'.

Ordering does not provide any advantages for random or ordered access of the records based on values of the other *nonordering fields* of the file. In these cases, we

do a linear search for random access. To access the records in order based on a non-ordering field, it is necessary to create another sorted copy—in a different order—of the file.

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time-consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

One option for making insertion more efficient is to keep some unused space in each block for new records. However, once this space is used up, the original problem resurfaces. Another frequently used method is to create a temporary *unordered* file called an **overflow** or **transaction** file. With this technique, the actual ordered file is called the **main** or **master** file. New records are inserted at the end of the overflow file rather than in their correct position in the main file. Periodically, the overflow file is sorted and merged with the master file during file reorganization. Insertion becomes very efficient, but at the cost of increased complexity in the search algorithm. One option is to keep the highest value of the key in each block in a separate field after taking into account the keys that have overflown from that block. Otherwise, the overflow file must be searched using a linear search if, after the binary search, the record is not found in the main file. For applications that do not require the most up-to-date information, overflow records can be ignored during a search.

Modifying a field value of a record depends on two factors: the search condition to locate the record and the field to be modified. If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search. A nonordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file. This requires deletion of the old record followed by insertion of the modified record.

Reading the file records in order of the ordering field is efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in this case, first we can reorganize the file, and then read its blocks sequentially. To reorganize the file, first we sort the records in the overflow file, and then merge them with the master file. The records marked for deletion are removed during the reorganization.

Table 16.3 summarizes the average access time in block accesses to find a specific record in a file with b blocks.

Ordered files are rarely used in database applications unless an additional access path, called a **primary index**, is used; this results in an **indexed-sequential file**.

Table 16.3 Average Access Times for a File of *b* Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

This further improves the random access time on the ordering key field. (We discuss indexes in Chapter 17.) If the ordering attribute is not a key, the file is called a **clustered file**.

16.8 Hashing Techniques

Another type of primary file organization is based on hashing, which provides very fast access to records under certain search conditions. This organization is usually called a **hash file**.[13] The search condition must be an equality condition on a single field, called the **hash field**. In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**. The idea behind hashing is to provide a function *h*, called a **hash function** or **randomizing function**, which is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field. We describe the use of hashing for internal files in Section 16.8.1; then we show how it is modified to store external files on disk in Section 16.8.2. In Section 16.8.3 we discuss techniques for extending hashing to dynamically growing files.

16.8.1 Internal Hashing

For internal files, hashing is typically implemented as a **hash table** through the use of an array of records. Suppose that the array index range is from 0 to $M - 1$, as shown in Figure 16.8(a); then we have M **slots** whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and $M - 1$. One common hash function is the $h(K) = K$ **mod** *M* function, which returns the remainder of an integer hash field value *K* after division by *M*; this value is then used for the record address.

---

[13]A hash file has also been called a *direct file*.

(a)

	Name	Ssn	Job	Salary
0				
1				
2				
3				
⋮				
$M-2$				
$M-1$				

Figure 16.8
Internal hashing data structures. (a) Array of M positions for use in internal hashing. (b) Collision resolution by chaining records.

	Data fields	Overflow pointer	
(b) 0		-1	
1		M	
2		-1	
3		-1	
4		$M+2$	
⋮			Address space
$M-2$		$M+1$	
$M-1$		-1	
M		$M+5$	
$M+1$		-1	
$M+2$		$M+4$	
⋮			Overflow space
$M+0-2$			
$M+0-1$			

- null pointer $= -1$
- overflow pointer refers to position of next record in linked list

Noninteger hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values. For a hash field whose data type is a string of 20 characters, Algorithm 16.2(a) can be used to calculate the hash address. We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type K: *array* [1..20] *of char* (in Pascal) or *char* $K[20]$ (in C).

Algorithm 16.2. Two simple hashing algorithms: (a) Applying the mod hash function to a character string K. (b) Collision resolution by open addressing.

(a) $temp \leftarrow 1$;
   for $i \leftarrow 1$ to 20 do $temp \leftarrow temp * \text{code}(K[i]) \bmod M$;
   $hash\_address \leftarrow temp \bmod M$;

(b) $i \leftarrow hash\_address(K)$; $a \leftarrow i$;
   if location i is occupied
       then **begin** $i \leftarrow (i + 1) \bmod M$;
           while $(i \neq a)$ and location i is occupied
               do $i \leftarrow (i + 1) \bmod M$;
           if $(i = a)$ then all positions are full
           else $new\_hash\_address \leftarrow i$;
           end;

Other hashing functions can be used. One technique, called **folding**, involves applying an arithmetic function such as *addition* or a logical function such as *exclusive or* to different portions of the hash field value to calculate the hash address (for example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address: (235+964) mod 1000 = 199). Another technique involves picking some digits of the hash field value—for instance, the third, fifth, and eighth digits—to form the hash address (for example, storing 1,000 employees with Social Security numbers of 10 digits into a hash file with 1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function).[14] The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because the **hash field space**—the number of possible values a hash field can take—is usually much larger than the **address space**—the number of available addresses for records. The hashing function maps the hash field space to the address space.

A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called **collision resolution**. There are numerous methods for collision resolution, including the following:

■ **Open addressing.** Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found. Algorithm 16.2(b) may be used for this purpose.

■ **Chaining.** For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. Additionally, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.

---

[14] A detailed discussion of hashing functions is outside the scope of our presentation.

A linked list of overflow records for each hash address is thus maintained, as shown in Figure 16.8(b).

■ **Multiple hashing.** The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary. Note that the series of hash functions are used in the same order for retrieval.

Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The algorithms for chaining are the simplest. Deletion algorithms for open addressing are rather tricky. Data structures textbooks discuss internal hashing algorithms in more detail.
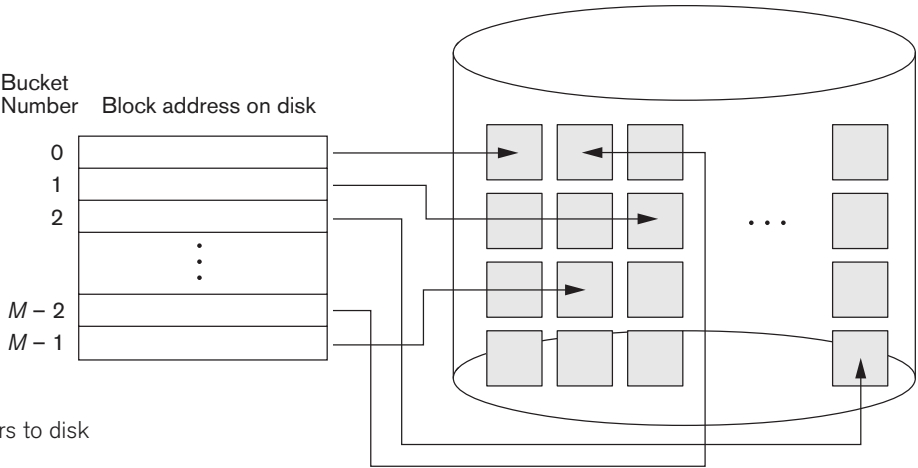
The goal of a good hashing function is twofold: first, to distribute the records uniformly over the address space so as to minimize collisions, thus making it possible to locate a record with a given key in a single access. The second, somewhat conflicting, goal is to achieve the above yet occupy the buckets fully, thus not leaving many unused locations. Simulation and analysis studies have shown that it is usually best to keep a hash file between 70 and 90% full so that the number of collisions remains low and we do not waste too much space. Hence, if we expect to have r records to store in the table, we should choose M locations for the address space such that (r/M) is between 0.7 and 0.9. It may also be useful to choose a prime number for M, since it has been demonstrated that this distributes the hash addresses better over the address space when the mod hashing function is used modulo a prime number. Other hash functions may require M to be a power of 2.
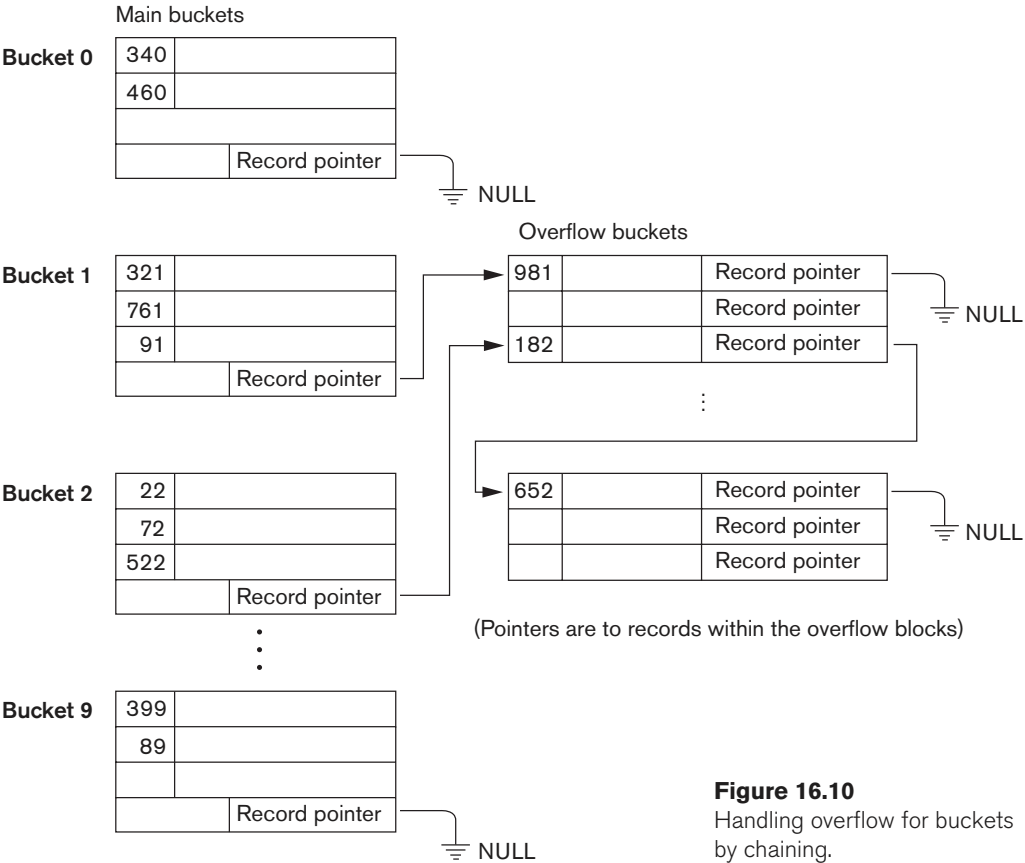
16.8.2 External Hashing for Disk Files

Hashing for disk files is called **external hashing**. To suit the characteristics of disk storage, the target address space is made of **buckets**, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function maps a key into a relative bucket number rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in Figure 16.9.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket. We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket, as shown in Figure 16.10. The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block.

Hashing provides the fastest possible access for retrieving an arbitrary record given the value of its hash field. Although most good hash functions do not maintain

Bucket
Number   Block address on disk

Figure 16.9
Matching bucket numbers to disk
block addresses.

(Pointers are to records within the overflow blocks)

Figure 16.10
Handling overflow for buckets
by chaining.

records in order of hash field values, some functions—called **order preserving**—do. A simple example of an order-preserving hash function is to take the leftmost three digits of an invoice number field that yields a bucket address as the hash address and keep the records sorted by invoice number within each bucket. Another example is to use an integer hash key directly as an index to a relative file, if the hash key values fill up a particular interval; for example, if employee numbers in a company are assigned as 1, 2, 3, … up to the total number of employees, we can use the identity hash function (i.e., Relative Address = Key) that maintains order. Unfortunately, this only works if sequence keys are generated in order by some application.

The hashing scheme described so far is called **static hashing** because a fixed number of buckets M is allocated. The function does key-to-address mapping, whereby we are fixing the address space. This can be a serious drawback for dynamic files. Suppose that we allocate M buckets for the address space and let m be the maximum number of records that can fit in one bucket; then at most $(m * M)$ records will fit in the allocated space. If the number of records turns out to be substantially fewer than $(m * M)$, we are left with a lot of unused space. On the other hand, if the number of records increases to substantially more than $(m * M)$, numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records. In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These reorganizations can be quite time-consuming for large files. Newer dynamic file organizations based on hashing allow the number of buckets to vary dynamically with only localized reorganization (see Section 16.8.3).

When using external hashing, searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file. Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in overflow, we simply remove it from the linked list. Notice that removing an overflow record implies that we should keep track of empty positions in overflow. This is done easily by maintaining a linked list of unused overflow locations.

Modifying a specific record's field value depends on two factors: the search condition to locate that specific record and the field to be modified. If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hashing function; otherwise, we must do a linear search. A nonhash field can be modified by changing the record and rewriting it in the same bucket. Modifying the hash field means that the record can move to another bucket, which requires deletion of the old record followed by insertion of the modified record.

16.8.3 Hashing Techniques That Allow Dynamic File Expansion

A major drawback of the *static* hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The schemes described in this section attempt to remedy this situation. The first

scheme—extendible hashing—stores an access structure in addition to the file, and hence is somewhat similar to indexing (see Chapter 17). The main difference is that the access structure is based on the values that result after application of the hash function to the search field. In indexing, the access structure is based on the values of the search field itself. The second technique, called linear hashing, does not require additional access structures. Another scheme, called **dynamic hashing**, uses an access structure based on binary tree data structures.
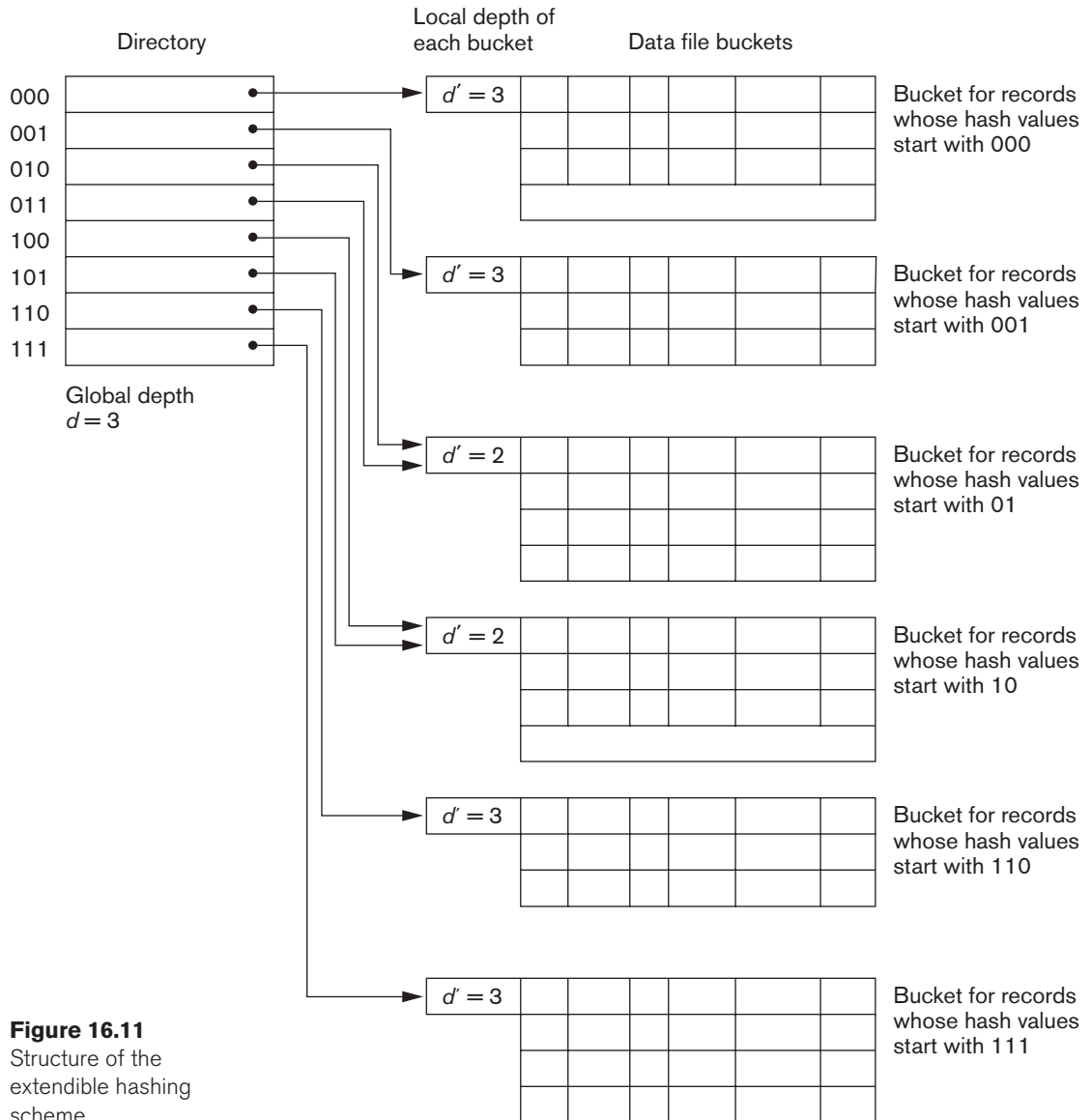
These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number. The access structure is built on the **binary representation** of the hashing function result, which is a string of **bits**. We call this the **hash value** of a record. Records are distributed among buckets based on the values of the *leading bits* in their hash values.

Extendible Hashing. In extendible hashing, proposed by Fagin (1979), a type of directory—an array of 2^d bucket addresses—is maintained, where d is called the **global depth** of the directory. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored. However, there does not have to be a distinct bucket for each of the 2^d directory locations. Several directory locations with the same first d' bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A **local depth** d'—stored with each bucket—specifies the number of bits on which the bucket contents are based. Figure 16.11 shows a directory with global depth $d = 3$.

The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket, whose local depth d' is equal to the global depth d, overflows. Halving occurs if $d > d'$ for all the buckets after some deletions occur. Most record retrievals require two block accesses—one to the directory and the other to the bucket.

To illustrate bucket splitting, suppose that a new inserted record causes overflow in the bucket whose hash values start with 01—the third bucket in Figure 16.11. The records will be distributed between two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth d' of the two new buckets is 3, which is one more than the local depth of the old bucket.

If a bucket that overflows and is split used to have a local depth d' equal to the global depth d of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 in Figure 16.11 overflows, the two new buckets need a directory with global depth $d = 4$, because the two buckets are now labeled 1110 and 1111, and hence their local depths are both 4. The directory size is hence doubled, and each of the other original locations in the

Figure 16.11
Structure of the extendible hashing scheme.

directory is also split into two locations, both of which have the same pointer value as did the original location.

The main advantage of extendible hashing that makes it attractive is that the *performance of the file does not degrade as the file grows*, as opposed to static external hashing, where collisions increase and the corresponding chaining effectively increases the average number of accesses per key. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated
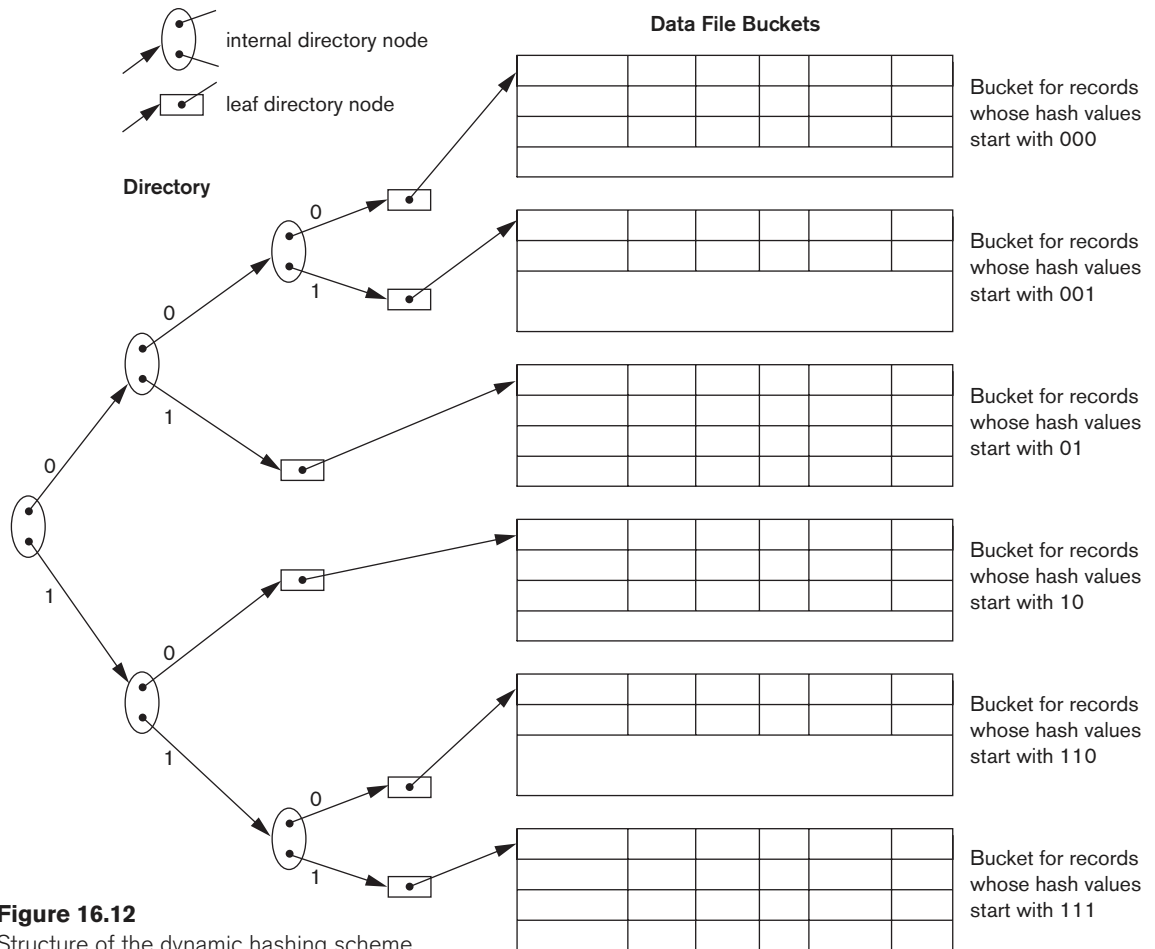
dynamically as needed. The space overhead for the directory table is negligible. The maximum directory size is 2^k, where k is the number of bits in the hash value. Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and thus the scheme is considered quite desirable for dynamic files.

Dynamic Hashing. A precursor to extendible hashing was dynamic hashing proposed by Larson (1978), in which the addresses of the buckets were either the n high-order bits or $n − 1$ high-order bits, depending on the total number of keys belonging to the respective bucket. The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing. The major difference is in the organization of the directory. Whereas extendible hashing uses the notion of global depth (high-order d bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth $d − 1$, dynamic hashing maintains a tree-structured directory with two types of nodes:

- Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.
- Leaf nodes—these hold a pointer to the actual bucket with records.

An example of the dynamic hashing appears in Figure 16.12. Four buckets are shown ("000", "001", "110", and "111") with high-order 3-bit addresses (corresponding to the global depth of 3), and two buckets ("01" and "10") are shown with high-order 2-bit addresses (corresponding to the local depth of 2). The latter two are the result of collapsing the "010" and "011" into "01" and collapsing "100" and "101" into "10". Note that the directory nodes are used implicitly to determine the "global" and "local" depths of buckets in dynamic hashing. The search for a record given the hashed address involves traversing the directory tree, which leads to the bucket holding that record. It is left to the reader to develop algorithms for insertion, deletion, and searching of records for the dynamic hashing scheme.

Linear Hashing. The idea behind linear hashing, proposed by Litwin (1980), is to allow a hash file to expand and shrink its number of buckets dynamically *without* needing a directory. Suppose that the file starts with M buckets numbered 0, 1, … , $M − 1$ and uses the mod hash function $h(K) = K \bmod M$; this hash function is called the **initial hash function** h_i. Overflow because of collisions is still needed and can be handled by maintaining individual overflow chains for each bucket. However, when a collision leads to an overflow record in *any* file bucket, the *first* bucket in the file—bucket 0—is split into two buckets: the original bucket 0 and a new bucket M at the end of the file. The records originally in bucket 0 are distributed between the two buckets based on a different hashing function $h_{i+1}(K) = K \bmod 2M$. A key property of the two hash functions h_i and h_{i+1} is that any records that hashed to bucket 0

Figure 16.12
Structure of the dynamic hashing scheme.

based on h_i will hash to either bucket 0 or bucket M based on h_{i+1}; this is necessary for linear hashing to work.

As further collisions lead to overflow records, additional buckets are split in the *linear* order 1, 2, 3, … . If enough overflows occur, all the original file buckets 0, 1, … , $M − 1$ will have been split, so the file now has $2M$ instead of M buckets, and all buckets use the hash function h_{i+1}. Hence, the records in overflow are eventually redistributed into regular buckets, using the function h_{i+1} via a *delayed split* of their buckets. There is no directory; only a value n—which is initially set to 0 and is incremented by 1 whenever a split occurs—is needed to determine which buckets have been split. To retrieve a record with hash key value K, first apply the function h_i to K; if $h_i(K) < n$, then apply the function h_{i+1} on K because the bucket is already split. Initially, $n = 0$, indicating that the function h_i applies to all buckets; n grows linearly as buckets are split.

When $n = M$ after being incremented, this signifies that all the original buckets have been split and the hash function h_{i+1} applies to all records in the file. At this point, n is reset to 0 (zero), and any new collisions that cause overflow lead to the use of a new hashing function $h_{i+2}(K) = K \bmod 4M$. In general, a sequence of hashing functions $h_{i+j}(K) = K \bmod (2^j M)$ is used, where $j = 0, 1, 2, \ldots$; a new hashing function h_{i+j+1} is needed whenever all the buckets $0, 1, \ldots, (2^j M) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 16.3.

Splitting can be controlled by monitoring the file load factor instead of by splitting whenever an overflow occurs. In general, the **file load factor** l can be defined as $l = r/(bfr * N)$, where r is the current number of file records, bfr is the maximum number of records that can fit in a bucket, and N is the current number of file buckets. Buckets that have been split can also be recombined if the load factor of the file falls below a certain threshold. Blocks are combined linearly, and N is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range. Splits can be triggered when the load exceeds a certain threshold—say, 0.9—and combinations can be triggered when the load falls below another threshold—say, 0.7. The main advantages of linear hashing are that it maintains the load factor fairly constantly while the file grows and shrinks, and it does not require a directory.[15]

> **Algorithm 16.3.** The Search Procedure for Linear Hashing
> if $n = 0$
>   then $m \leftarrow h_j (K)$ (*m is the hash value of record with hash key K*)
>   else **begin**
> $m \leftarrow h_j (K)$;
>   if $m < n$ then $m \leftarrow h_{j+1} (K)$
>   **end**;

search the bucket whose hash value is m (and its overflow, if any);

16.9 Other Primary File Organizations

16.9.1 Files of Mixed Records

The file organizations we have studied so far assume that all records of a particular file are of the same record type. The records could be of EMPLOYEEs, PROJECTs, STUDENTs, or DEPARTMENTs, but each file contains records of only one type. In most database applications, we encounter situations in which numerous types of entities are interrelated in various ways, as we saw in Chapter 7. Relationships among records in various files can be represented by **connecting fields**.[16] For example, a

---

[15]For details of insertion and deletion into Linear hashed files, refer to Litwin (1980) and Salzberg (1988).

[16]The concept of foreign keys in the relational data model (Chapter 3) and references among objects in object-oriented models (Chapter 11) are examples of connecting fields.

STUDENT record can have a connecting field Major_dept whose value gives the name of the DEPARTMENT in which the student is majoring. This Major_dept field *refers* to a DEPARTMENT entity, which should be represented by a record of its own in the DEPARTMENT file. If we want to retrieve field values from two related records, we must retrieve one of the records first. Then we can use its connecting field value to retrieve the related record in the other file. Hence, relationships are implemented by **logical field references** among the records in distinct files.

File organizations in object DBMSs, as well as legacy systems such as hierarchical and network DBMSs, often implement relationships among records as **physical relationships** realized by physical contiguity (or clustering) of related records or by physical pointers. These file organizations typically assign an **area** of the disk to hold records of more than one type so that records of different types can be **physically clustered** on disk. If a particular relationship is expected to be used frequently, implementing the relationship physically can increase the system's efficiency at retrieving related records. For example, if the query to retrieve a DEPARTMENT record and all records for STUDENTs majoring in that department is frequent, it would be desirable to place each DEPARTMENT record and its cluster of STUDENT records contiguously on disk in a mixed file. The concept of **physical clustering** of object types is used in object DBMSs to store related objects together in a mixed file. In data warehouses (see Chapter 29), the input data comes from a variety of sources and undergoes an integration initially to collect the required data into an **operational data store (ODS)**. An ODS typically contains files where records of multiple types are kept together. It is passed on to a data warehouse after **ETL** (extract, transform and load) processing operations are performed on it.

To distinguish the records in a mixed file, each record has—in addition to its field values—a **record type** field, which specifies the type of record. This is typically the first field in each record and is used by the system software to determine the type of record it is about to process. Using the catalog information, the DBMS can determine the fields of that record type and their sizes, in order to interpret the data values in the record.

16.9.2 B-Trees and Other Data Structures as Primary Organization

Other data structures can be used for primary file organizations. For example, if both the record size and the number of records in a file are small, some DBMSs offer the option of a B-tree data structure as the primary file organization. We will describe B-trees in Section 17.3.1, when we discuss the use of the B-tree data structure for indexing. In general, any data structure that can be adapted to the characteristics of disk devices can be used as a primary file organization for record placement on disk. Recently, column-based storage of data has been proposed as a primary method for storage of relations in relational databases. We will briefly introduce it in Chapter 17 as a possible alternative storage scheme for relational databases.

16.10 Parallelizing Disk Access Using RAID Technology

With the exponential growth in the performance and capacity of semiconductor devices and memories, faster microprocessors with larger and larger primary memories are continually becoming available. To match this growth, it is natural to expect that secondary storage technology must also take steps to keep up with processor technology in performance and reliability.
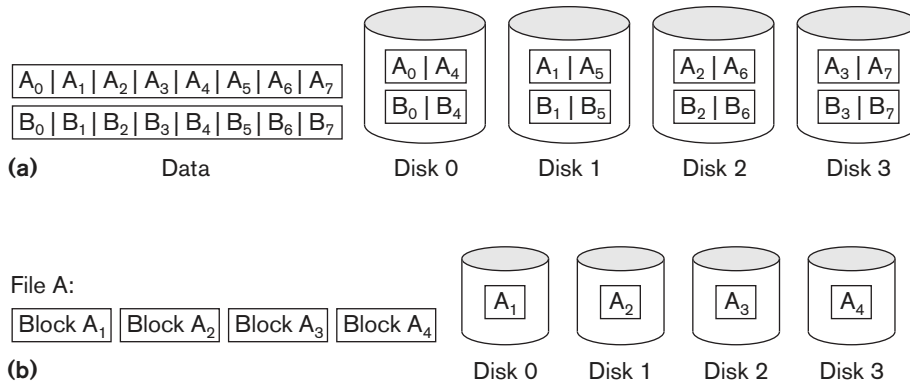
A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **redundant arrays of inexpensive disks**. More recently, the *I* in RAID is said to stand for *independent*. The RAID idea received a very positive industry endorsement and has been developed into an elaborate set of alternative RAID architectures (RAID levels 0 through 6). We highlight the main features of the technology in this section.

The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.[17] Although RAM capacities have quadrupled every two to three years, disk *access times* are improving at less than 10% per year, and disk *transfer rates* are improving at roughly 20% per year. Disk *capacities* are indeed improving at more than 50% per year, but the speed and access time improvements are of a much smaller magnitude.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving video, audio, image, and spatial data processing (see Chapters 26 for details of these applications), with corresponding lack of fast access to large, shared data sets.

The natural solution is a large array of small independent disks acting as a single higher performance logical disk. A concept called data striping is used, which utilizes *parallelism* to improve disk performance. **Data striping** distributes data transparently over multiple disks to make them appear as a single large, fast disk. Figure 16.13 shows a file distributed or *striped* over four disks. In **bit-level striping**, a byte is split and individual bits are stored on independent disks. Figure 16.13(a) illustrates bit-striping across four disks where the bits (0, 4) are assigned to disk 0, bits (1, 5) to disk 1, and so on. With this striping, every disk participates in every read or write operation; the number of accesses per second would remain the same as on a single disk, but the amount of data read in a given time would increase fourfold. Thus, striping improves overall I/O performance by providing high overall transfer rates. **Block-level striping** stripes blocks across disks. It treats the array of disks as if it is one disk. Blocks are logically numbered from 0 in sequence. Disks in an *m*-disk array are numbered 0 to *m* – 1. With striping, block j goes to disk (j mod *m*). Figure 16.13(b) illustrates block striping with four disks (*m* = 4). Data striping also accomplishes load balancing among disks. Moreover, by storing redundant information on

---

[17]This was predicted by Gordon Bell to be about 40% every year between 1974 and 1984 and is now supposed to exceed 50% per year.

(a)   Data   Disk 0   Disk 1   Disk 2   Disk 3

(b)   Disk 0   Disk 1   Disk 2   Disk 3

Figure 16.13
Striping of data
across multiple disks.
(a) Bit-level striping
across four disks.
(b) Block-level striping
across four disks.

disks using parity or some other error-correction code, reliability can be improved. In Sections 16.10.1 and 16.10.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 16.10.3 discusses RAID organizations and levels.

16.10.1 Improving Reliability with RAID

For an array of n disks, the likelihood of failure is n times as much as that for one disk. Hence, if the MTBF (mean time between failures) of a disk drive is assumed to be 200,000 hours or about 22.8 years (for the disk drive in Table 16.1 called Seagate Enterprise Performance 10K HDD, it is 1.4 million hours), the MTBF for a bank of 100 disk drives becomes only 2,000 hours or 83.3 days (for a bank of 1,000 Seagate Enterprise Performance 10K HDD disks it would be 1,400 hours or 58.33 days). Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

One technique for introducing redundancy is called **mirroring** or **shadowing**. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours; then the mean time to data loss of a mirrored disk system using 100 disks with MTBF of 200,000 hours each is $(200,000)^2/(2 * 24) = 8.33 * 10^8$ hours, which is 95,028 years.[18] Disk mirroring also doubles the rate at which read requests are handled, since a read can go to either disk. The transfer rate of each read, however, remains the same as that for a single disk.

---

[18]The formulas for MTBF calculations appear in Chen et al. (1994).

Another solution to the problem of reliability is to store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure. The incorporation of redundancy must consider two problems: selecting a technique for computing the redundant information, and selecting a method of distributing the redundant information across the disk array. The first problem is addressed by using error-correcting codes involving parity bits, or specialized codes such as Hamming codes. Under the parity scheme, a redundant disk may be considered as having the sum of all the data in the other disks. When a disk fails, the missing information can be constructed by a process similar to subtraction.

For the second problem, the two major approaches are either to store the redundant information on a small number of disks or to distribute it uniformly across all disks. The latter results in better load balancing. The different levels of RAID choose a combination of these options to implement redundancy and improve reliability.

16.10.2 Improving Performance with RAID

The disk arrays employ the technique of data striping to achieve higher transfer rates. Note that data can be read or written only one block at a time, so a typical transfer contains 512 to 8,192 bytes. Disk striping may be applied at a finer granularity by breaking up a byte of data into bits and spreading the bits to different disks. Thus, **bit-level data striping** consists of splitting a byte of data and writing bit j to the jth disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate. Each disk participates in each I/O request and the total amount of data read per request is eight times as much. Bit-level striping can be generalized to a number of disks that is either a multiple or a factor of eight. Thus, in a four-disk array, bit n goes to the disk which is (n mod 4). Figure 16.13(a) shows bit-level striping of data.

The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to **block-level striping**. Figure 16.13(b) shows block-level data striping assuming the data file contains four blocks. With block-level striping, multiple independent requests that access single blocks (small requests) can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests. Requests that access multiple blocks (large requests) can be parallelized, thus reducing their response time. In general, the more the number of disks in an array, the larger the potential performance benefit. However, assuming independent failures, the disk array of 100 disks collectively has 1/100th the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.
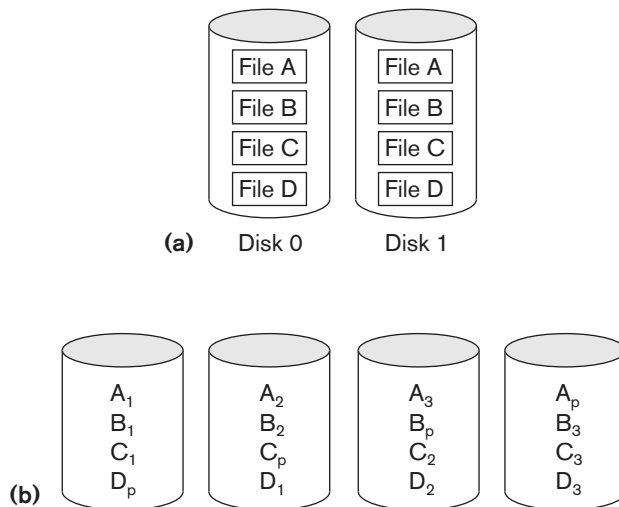
16.10.3 RAID Organizations and Levels

Different RAID organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. In the initial proposal, levels 1 through 5 of RAID were proposed, and two additional levels—0 and 6—were added later.

RAID level 0 uses data striping, has no redundant data, and hence has the best write performance since updates do not have to be duplicated. It splits data evenly across two or more disks. However, its read performance is not as good as RAID level 1, which uses mirrored disks. In the latter, performance improvement is possible by scheduling a read request to the disk with shortest expected seek and rotational delay. RAID level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Thus, in one particular version of this level, three redundant disks suffice for four original disks, whereas with mirroring—as in level 1—four would be required. Level 2 includes both error detection and correction, although detection is generally not required because broken disks identify themselves.

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed. Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Figure 16.14(b) shows an illustration of RAID level 5, where parity is shown with subscript p. If one disk fails, the missing data is calculated based on the parity available from the remaining disks. Finally, RAID level 6 applies the so-called $P + Q$ redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

Rebuilding in case of disk failure is easiest for RAID level 1. Other levels require the reconstruction of a failed disk by reading multiple disks. Level 1 is used for critical applications such as storing logs of transactions. Levels 3 and 5 are preferred for large volume storage, with level 3 providing higher transfer rates. Most popular use of RAID technology currently uses level 0 (with striping), level 1 (with mirroring), and level 5 with an extra drive for parity. A combination of multiple RAID levels are also used—for example, 0 + 1 combines striping and mirroring

(a)   Disk 0   Disk 1

(b)

Figure 16.14
Some popular levels of RAID. (a) RAID level 1: Mirroring of data on two disks. (b) RAID level 5: Striping of data with distributed parity across four disks.

using a minimum of four disks. Other nonstandard RAID levels include: RAID 1.5, RAID 7, RAID-DP, RAID S or Parity RAID, Matrix RAID, RAID-K, RAID-Z, RAIDn, Linux MD RAID 10, IBM ServeRAID 1E, and unRAID. A discussion of these nonstandard levels is beyond the scope of this text. Designers of a RAID setup for a given application mix have to confront many design decisions such as the level of RAID, the number of disks, the choice of parity schemes, and grouping of disks for block-level striping. Detailed performance studies on small reads and writes (referring to I/O requests for one striping unit) and large reads and writes (referring to I/O requests for one stripe unit from each disk in an error-correction group) have been performed.

16.11 Modern Storage Architectures

In this section, we describe some recent developments in storage systems that are becoming an integral part of most enterprise's information system architectures. We already mentioned the SATA and SAS interface, which has almost replaced the previously popular SCSI (small computer system interface) in laptops and small servers. The Fibre Channel (FC) interface is the predominant choice for storage networks in data centers. We review some of the modern storage architectures next.

16.11.1 Storage Area Networks

With the rapid growth of electronic commerce, enterprise resource planning (ERP) systems that integrate application data across organizations, and data warehouses that keep historical aggregate information (see Chapter 29), the demand for storage has gone up substantially. For today's Internet-driven organizations, it has become necessary to move from a static fixed data center-oriented operation to a more flexible and dynamic infrastructure for the organizations' information processing requirements. The total cost of managing all data is growing so rapidly that in many instances the cost of managing server-attached storage exceeds the cost of the server itself. Furthermore, the procurement cost of storage is only a small fraction—typically, only 10 to 15% of the overall cost of storage management. Many users of RAID systems cannot use the capacity effectively because it has to be attached in a fixed manner to one or more servers. Therefore, most large organizations have moved to a concept called **storage area networks (SANs)**. In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.

Several companies have emerged as SAN providers and supply their own proprietary topologies. They allow storage systems to be placed at longer distances from the servers and provide different performance and connectivity options. Existing storage management applications can be ported into SAN configurations using Fibre Channel networks that encapsulate the legacy SCSI protocol. As a result, the SAN-attached devices appear as SCSI devices.

Current architectural alternatives for SAN include the following: point-to-point connections between servers and storage systems via Fiber Channel; use of a Fiber

Channel switch to connect multiple RAID systems, tape libraries, and so on to servers; and the use of Fiber Channel hubs and switches to connect servers and storage systems in different configurations. Organizations can slowly move up from simpler topologies to more complex ones by adding servers and storage devices as needed. We do not provide further details here because they vary among SAN vendors. The main advantages claimed include:

- Flexible many-to-many connectivity among servers and storage devices using Fiber Channel hubs and switches
- Up to 10 km separation between a server and a storage system using appropriate fiber optic cables
- Better isolation capabilities allowing nondisruptive addition of new peripherals and servers
- High-speed data replication across multiple storage systems. Typical technologies use synchronous replication for local and asynchronous replication for disaster recovery (DR) solutions.

SANs are growing very rapidly but are still faced with many problems, such as combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware. Most major companies are evaluating SANs as a viable option for database storage.

16.11.2 Network-Attached Storage

With the phenomenal growth in digital data, particularly generated from multimedia and other enterprise applications, the need for high-performance storage solutions at low cost has become extremely important. **Network-attached storage** (NAS) devices are among the storage devices being used for this purpose. These devices are, in fact, servers that do not provide any of the common server services, but simply allow the addition of storage for **file sharing**. NAS devices allow vast amounts of hard-disk storage space to be added to a network and can make that space available to multiple servers without shutting them down for maintenance and upgrades. NAS devices can reside anywhere on a local area network (LAN) and may be combined in different configurations. A single hardware device, often called the **NAS box** or **NAS head**, acts as the interface between the NAS system and network clients. These NAS devices require no monitor, keyboard, or mouse. One or more disk or tape drives can be attached to many NAS systems to increase total capacity. Clients connect to the NAS head rather than to the individual storage devices. A NAS can store any data that appears in the form of files, such as e-mail boxes, Web content, remote system backups, and so on. In that sense, NAS devices are being deployed as a replacement for traditional file servers.

NAS systems strive for reliable operation and easy administration. They include built-in features such as secure authentication, or the automatic sending of e-mail alerts in case of error on the device. The NAS devices (or *appliances*, as some vendors refer to them) are being offered with a high degree of scalability, reliability,

flexibility, and performance. Such devices typically support RAID levels 0, 1, and 5. Traditional storage area networks (SANs) differ from NAS in several ways. Specifically, SANs often utilize Fibre Channel rather than Ethernet, and a SAN often incorporates multiple network devices or *endpoints* on a self-contained or *private* LAN, whereas NAS relies on individual devices connected directly to the existing public LAN. Whereas Windows, UNIX, and NetWare file servers each demand specific protocol support on the client side, NAS systems claim greater operating system independence of clients. In summary, NAS provides a file system interface with support for networked files using protocols such as common internet file system (CIFS) or network file system (NFS).

16.11.3 iSCSI and Other Network-Based Storage Protocols

A new protocol called **iSCSI** (Internet SCSI) has been proposed recently. It is a block-storage protocol like SAN. It allows clients (called *initiators*) to send SCSI commands to SCSI storage devices on remote channels. The main advantage of iSCSI is that it does not require the special cabling needed by Fibre Channel and it can run over longer distances using existing network infrastructure. By carrying SCSI commands over IP networks, iSCSI facilitates data transfers over intranets and manages storage over long distances. It can transfer data over local area networks (LANs), wide area networks (WANs), or the Internet.

iSCSI works as follows. When a DBMS needs to access data, the operating system generates the appropriate SCSI commands and data request, which then go through encapsulation and, if necessary, encryption procedures. A packet header is added before the resulting IP packets are transmitted over an Ethernet connection. When a packet is received, it is decrypted (if it was encrypted before transmission) and disassembled, separating the SCSI commands and request. The SCSI commands go via the SCSI controller to the SCSI storage device. Because iSCSI is bidirectional, the protocol can also be used to return data in response to the original request. Cisco and IBM have marketed switches and routers based on this technology.

 iSCSI storage has mainly impacted small- and medium-sized businesses because of its combination of simplicity, low cost, and the functionality of iSCSI devices. It allows them not to learn the ins and outs of Fibre Channel (FC) technology and instead benefit from their familiarity with the IP protocol and Ethernet hardware. iSCSI implementations in the data centers of very large enterprise businesses are slow in development due to their prior investment in Fibre Channel–based SANs.

iSCSI is one of two main approaches to storage data transmission over IP networks. The other method, **Fibre Channel over IP (FCIP)**, translates Fibre Channel control codes and data into IP packets for transmission between geographically distant Fibre Channel storage area networks. This protocol, known also as *Fibre Channel tunneling* or *storage tunneling*, can only be used in conjunction with Fibre Channel technology, whereas iSCSI can run over existing Ethernet networks.

The latest idea to enter the enterprise IP storage race is **Fibre Channel over Ethernet (FCoE)**, which can be thought of as iSCSI without the IP. It uses many

elements of SCSI and FC (just like iSCSI), but it does not include TCP/IP components. FCoE has been successfully productized by CISCO (termed "Data Center Ethernet") and Brocade. It takes advantage of a reliable ethernet technology that uses buffering and end-to-end flow control to avoid dropped packets. This promises excellent performance, especially on 10 Gigabit Ethernet (10GbE), and is relatively easy for vendors to add to their products.

16.11.4 Automated Storage Tiering

Another trend in storage is automated storage tiering (AST), which automatically moves data between different storage types such as SATA, SAS, and solid-state drives (SSDs) depending on the need. The storage administrator can set up a tiering policy in which less frequently used data is moved to slower and cheaper SATA drives and more frequently used data is moved up to solid-state drives (see Table 16.1 for the various tiers of storage ordered by increasing speed of access). This automated tiering can improve database performance tremendously.

EMC has an implementation of this technology called FAST (fully automated storage tiering) that does continuous monitoring of data activity and takes actions to move the data to the appropriate tier based on the policy.

16.11.5 Object-Based Storage

During the last few years, there have been major developments in terms of rapid growth of the cloud concept, distributed architectures for databases and for analytics, and development of data-intensive applications on the Web (see Chapters 23, 24, and 25). These developments have caused fundamental changes in enterprise storage infrastructure. The hardware-oriented file-based systems are evolving into new open-ended architectures for storage. The latest among these is object-based storage. Under this scheme, data is managed in the form of objects rather than files made of blocks. Objects carry metadata that contains properties that can be used for managing those objects. Each object carries a unique global identifier that is used to locate it. Object storage has its origins in research projects at CMU (Gibson et al., 1996) on scaling up of network attached storage and in the Oceanstore system at UC Berkeley (Kubiatowicz et al., 2000), which attempted to build a global infrastructure over all forms of trusted and untrusted servers for continuous access to persistent data. There is no need to do lower level storage operations in terms of capacity management or making decisions like what type of RAID architecture should be used for fault protection.

Object storage also allows additional flexibility in terms of interfaces—it gives control to applications that can control the objects directly and also allows the objects to be addressable across a wide namespace spanning multiple devices. Replication and distribution of objects is also supported. In general, object storage is ideally suited for scalable storage of massive amounts of unstructured data such as Web pages, images, and audio/video clips and files. Object-based storage device commands (OSDs) were proposed as part of SCSI protocol a long time ago but did not

become a commercial product until Seagate adopted OSDs in its Kinetic Open Storage Platform. Currently, Facebook uses an object storage system to store photos at the level of over 350 Petabytes of storage; Spotify uses an object storage system for storing songs; and Dropbox uses it for its storage infrastructure. Object storage is the choice of many cloud offerings, such as Amazon's AWS (Amazon Web Service) S3, and Microsoft's Azure, which stores files, relations, messages, and so on as objects. Other examples of products include Hitachi's HCP, EMC's Atmos, and Scality's RING. Openstack Swift is an open source project that allows one to use HTTP GET and PUT to retrieve and store objects—that's basically the whole API. Openstack Swift uses very cheap hardware, is fully fault resistant, automatically takes advantage of geographic redundancy, and scales to very large numbers of objects. Since object storage forces locking to occur at the object level, it is not clear how suitable it is for concurrent transaction processing in high-throughput transaction-oriented systems. Therefore, it is still not considered viable for mainstream enterprise-level database applications.

16.12 Summary

We began this chapter by discussing the characteristics of memory hierarchies and then concentrated on secondary storage devices. In particular, we focused on magnetic disks because they are still the preferred medium to store online database files. Table 16.1 presented a perspective on the memory hierarchies and their current capacities, access speeds, transfer rates, and costs.

Data on disk is stored in blocks; accessing a disk block is expensive because of the seek time, rotational delay, and block transfer time. To reduce the average block access time, double buffering can be used when accessing consecutive disk blocks. (Other disk parameters are discussed in Appendix B.) We introduced the various interface technologies in use today for disk drives and optical devices. We presented a list of strategies employed to improve access of data from disks. We also introduced solid-state drives, which are rapidly becoming popular, and optical drives, which are mainly used as tertiary storage. We discussed the working of the buffer manager, which is responsible for handling data requests and we presented various buffer replacement policies. We presented different ways of storing file records on disk. File records are grouped into disk blocks and can be fixed length or variable length, spanned or unspanned, and of the same record type or mixed types. We discussed the file header, which describes the record formats and keeps track of the disk addresses of the file blocks. Information in the file header is used by system software accessing the file records.

Then we presented a set of typical commands for accessing individual file records and discussed the concept of the current record of a file. We discussed how complex record search conditions are transformed into simple search conditions that are used to locate records in the file.

Three primary file organizations were then discussed: unordered, ordered, and hashed. Unordered files require a linear search to locate records, but record

insertion is very simple. We discussed the deletion problem and the use of deletion markers.

Ordered files shorten the time required to read records in order of the ordering field. The time required to search for an arbitrary record, given the value of its ordering key field, is also reduced if a binary search is used. However, maintaining the records in order makes insertion very expensive; thus the technique of using an unordered overflow file to reduce the cost of record insertion was discussed. Overflow records are merged with the master file periodically, and deleted records are physically dropped during file reorganization.

Hashing provides very fast access to an arbitrary record of a file, given the value of its hash key. The most suitable method for external hashing is the bucket technique, with one or more contiguous blocks corresponding to each bucket. Collisions causing bucket overflow are handled by open addressing, chaining, or multiple hashing. Access on any nonhash field is slow, and so is ordered access of the records on any field. We discussed three hashing techniques for files that grow and shrink in the number of records dynamically: extendible, dynamic, and linear hashing. The first two use the higher-order bits of the hash address to organize a directory. Linear hashing is geared to keep the load factor of the file within a given range and adds new buckets linearly.

We briefly discussed other possibilities for primary file storage and organization, such as B-trees, and files of mixed records, which implement relationships among records of different types physically as part of the storage structure. We reviewed the recent advances in disk technology represented by RAID (redundant arrays of inexpensive (or independent) disks), which has become a standard technique in large enterprises to provide better reliability and fault tolerance features in storage. Finally, we reviewed some modern trends in enterprise storage systems: storage area networks (SANs), network-attached storage (NAS), iSCSI and other network based protocols, automatic storage tiering, and finally object-based storage, which is playing a major role in storage architecture of data centers offering cloud-based services.

Review Questions

16.1. What is the difference between primary and secondary storage?

16.2. Why are disks, not tapes, used to store online database files?

16.3. Define the following terms: *disk, disk pack, track, block, cylinder, sector, interblock gap, and read/write head.*

16.4. Discuss the process of disk initialization.

16.5. Discuss the mechanism used to read data from or write data to the disk.

16.6. What are the components of a disk block address?

16.7. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.

16.8. How does double buffering improve block access time?

16.9. What are the reasons for having variable-length records? What types of separator characters are needed for each?

16.10. Discuss the techniques for allocating file blocks on disk.

16.11. What is the difference between a file organization and an access method?

16.12. What is the difference between static and dynamic files?

16.13. What are the typical record-at-a-time operations for accessing a file? Which of these depend on the current file record?

16.14. Discuss the techniques for record deletion.

16.15. Discuss the advantages and disadvantages of using (a) an unordered file, (b) an ordered file, and (c) a static hash file with buckets and chaining. Which operations can be performed efficiently on each of these organizations, and which operations are expensive?

16.16. Discuss the techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?

16.17. What is the difference between the directories of extendible and dynamic hashing?

16.18. What are mixed files used for? What are other types of primary file organizations?

16.19. Describe the mismatch between processor and disk technologies.

16.20. What are the main goals of the RAID technology? How does it achieve them?

16.21. How does disk mirroring help improve reliability? Give a quantitative example.

16.22. What characterizes the levels in RAID organization?

16.23. What are the highlights of the popular RAID levels 0, 1, and 5?

16.24. What are storage area networks? What flexibility and advantages do they offer?

16.25. Describe the main features of network-attached storage as an enterprise storage solution.

16.26. How have new iSCSI systems improved the applicability of storage area networks?

16.27. What are SATA, SAS, and FC protocols?

16.28. What are solid-state drives (SSDs) and what advantage do they offer over HDDs?

16.29. What is the function of a buffer manager? What does it do to serve a request for data?

16.30. What are some of the commonly used buffer replacement strategies?

16.31. What are optical and tape jukeboxes? What are the different types of optical media served by optical drives?

16.32. What is automatic storage tiering? Why is it useful?

16.33. What is object-based storage? How is it superior to conventional storage systems?

Exercises

16.34. Consider a disk with the following characteristics (these are not parameters of any particular disk unit): block size B = 512 bytes; interblock gap size G = 128 bytes; number of blocks per track = 20; number of tracks per surface = 400. A disk pack consists of 15 double-sided disks.

a. What is the total capacity of a track, and what is its useful capacity (excluding interblock gaps)?

b. How many cylinders are there?

c. What are the total capacity and the useful capacity of a cylinder?

d. What are the total capacity and the useful capacity of a disk pack?

e. Suppose that the disk drive rotates the disk pack at a speed of 2,400 rpm (revolutions per minute); what are the transfer rate (tr) in bytes/msec and the block transfer time (btt) in msec? What is the average rotational delay (rd) in msec? What is the bulk transfer rate? (See Appendix B.)

f. Suppose that the average seek time is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block, given its block address?

g. Calculate the average time it would take to transfer 20 random blocks, and compare this with the time it would take to transfer 20 consecutive blocks using double buffering to save seek time and rotational delay.

16.35. A file has r = 20,000 STUDENT records of *fixed length*. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Address (40 bytes), PHONE (10 bytes), Birth_date (8 bytes), Sex (1 byte), Major_dept_code (4 bytes), Minor_dept_code (4 bytes), Class_code (4 bytes, integer), and Degree_program (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 16.27.

a. Calculate the record size R in bytes.

b. Calculate the blocking factor bfr and the number of file blocks b, assuming an unspanned organization.

    c. Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously, and double buffering is used; (ii) the file blocks are not stored contiguously.

    d. Assume that the file is ordered by Ssn; by doing a binary search, calculate the time it takes to search for a record given its Ssn value.

16.36. Suppose that only 80% of the STUDENT records from Exercise 16.28 have a value for Phone, 85% for Major_dept_code, 15% for Minor_dept_code, and 90% for Degree_program; and suppose that we use a variable-length record file. Each record has a 1-byte *field type* for each field in the record, plus the 1-byte deletion marker and a 1-byte end-of-record marker. Suppose that we use a *spanned* record organization, where each block has a 5-byte pointer to the next block (this space is not used for record storage).

    a. Calculate the average record length R in bytes.

    b. Calculate the number of blocks needed for the file.

16.37. Suppose that a disk unit has the following parameters: seek time $s = 20$ msec; rotational delay $rd = 10$ msec; block transfer time $btt = 1$ msec; block size $B = 2400$ bytes; interblock gap size $G = 600$ bytes. An EMPLOYEE file has the following fields: Ssn, 9 bytes; Last_name, 20 bytes; First_name, 20 bytes; Middle_init, 1 byte; Birth_date, 10 bytes; Address, 35 bytes; Phone, 12 bytes; Supervisor_ssn, 9 bytes; Department, 4 bytes; Job_code, 4 bytes; deletion marker, 1 byte. The EMPLOYEE file has $r = 30,000$ records, fixed-length format, and unspanned blocking. Write appropriate formulas *and* calculate the following values for the above EMPLOYEE file:

    a. Calculate the record size R (including the deletion marker), the blocking factor bfr, and the number of disk blocks b.

    b. Calculate the wasted space in each disk block because of the unspanned organization.

    c. Calculate the transfer rate tr and the bulk transfer rate btr for this disk unit (see Appendix B for definitions of tr and btr).

    d. Calculate the average *number of block accesses* needed to search for an arbitrary record in the file, using linear search.

    e. Calculate in msec the average *time* needed to search for an arbitrary record in the file, using linear search, if the file blocks are stored on consecutive disk blocks and double buffering is used.

    f. Calculate in msec the average *time* needed to search for an arbitrary record in the file, using linear search, if the file blocks are *not* stored on consecutive disk blocks.

    g. Assume that the records are ordered via some key field. Calculate the average *number of block accesses* and the *average time* needed to search for an arbitrary record in the file, using binary search.

16.38. A PARTS file with Part# as the hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428,

3943, 4750, 6975, 4981, and 9208. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order, using the hash function $h(K) = K \bmod 8$. Calculate the average number of block accesses for a random retrieval on Part#.

16.39. Load the records of Exercise 16.31 into expandable hash files based on extendible hashing. Show the structure of the directory at each step, and the global and local depths. Use the hash function $h(K) = K \bmod 128$.

16.40. Load the records of Exercise 16.31 into an expandable hash file, using linear hashing. Start with a single disk block, using the hash function $h_0 = K \bmod 2^0$, and show how the file grows and how the hash functions change as the records are inserted. Assume that blocks are split whenever an overflow occurs, and show the value of n at each stage.

16.41. Compare the file commands listed in Section 16.5 to those available on a file access method you are familiar with.

16.42. Suppose that we have an unordered file of fixed-length records that uses an unspanned record organization. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.

16.43. Suppose that we have an ordered file of fixed-length records and an unordered overflow file to handle insertion. Both files use unspanned records. Outline algorithms for insertion, deletion, and modification of a file record and for reorganizing the file. State any assumptions you make.

16.44. Can you think of techniques other than an unordered overflow file that can be used to make insertions in an ordered file more efficient?

16.45. Suppose that we have a hash file of fixed-length records, and suppose that overflow is handled by chaining. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.

16.46. Can you think of techniques other than chaining to handle bucket overflow in external hashing?

16.47. Write pseudocode for the insertion algorithms for linear hashing and for extendible hashing.

16.48. Write program code to access individual fields of records under each of the following circumstances. For each case, state the assumptions you make concerning pointers, separator characters, and so on. Determine the type of information needed in the file header in order for your code to be general in each case.
   a. Fixed-length records with unspanned blocking
   b. Fixed-length records with spanned blocking
   c. Variable-length records with variable-length fields and spanned blocking
   d. Variable-length records with repeating groups and spanned blocking
   e. Variable-length records with optional fields and spanned blocking
   f. Variable-length records that allow all three cases in parts c, d, and e

16.49. Suppose that a file initially contains $r = 120,000$ records of $R = 200$ bytes each in an unsorted (heap) file. The block size $B = 2,400$ bytes, the average seek time $s = 16$ ms, the average rotational latency $rd = 8.3$ ms, and the block transfer time $btt = 0.8$ ms. Assume that 1 record is deleted for every 2 records added until the total number of active records is 240,000.

   a. How many block transfers are needed to reorganize the file?

   b. How long does it take to find a record right before reorganization?

   c. How long does it take to find a record right after reorganization?

16.50. Suppose we have a sequential (ordered) file of 100,000 records where each record is 240 bytes. Assume that $B = 2,400$ bytes, $s = 16$ ms, $rd = 8.3$ ms, and $btt = 0.8$ ms. Suppose we want to make X independent random record reads from the file. We could make X random block reads or we could perform one exhaustive read of the entire file looking for those X records. The question is to decide when it would be more efficient to perform one exhaustive read of the entire file than to perform X individual random reads. That is, what is the value for X when an exhaustive read of the file is more efficient than random X reads? Develop this as a function of X.

16.51. Suppose that a static hash file initially has 600 buckets in the primary area and that records are inserted that create an overflow area of 600 buckets. If we reorganize the hash file, we can assume that most of the overflow is eliminated. If the cost of reorganizing the file is the cost of the bucket transfers (reading and writing all of the buckets) and the only periodic file operation is the fetch operation, then how many times would we have to perform a fetch (successfully) to make the reorganization cost effective? That is, the reorganization cost and subsequent search cost are less than the search cost before reorganization. Support your answer. Assume $s = 16$ msec, $rd = 8.3$ msec, and $btt = 1$ msec.

16.52. Suppose we want to create a linear hash file with a file load factor of 0.7 and a blocking factor of 20 records per bucket, which is to contain 112,000 records initially.

   a. How many buckets should we allocate in the primary area?

   b. What should be the number of bits used for bucket addresses?

Selected Bibliography

Wiederhold (1987) has a detailed discussion and analysis of secondary storage devices and file organizations as a part of database design. Optical disks are described in Berg and Roth (1989) and analyzed in Ford and Christodoulakis (1991). Flash memory is discussed by Dipert and Levy (1993). Ruemmler and Wilkes (1994) present a survey of the magnetic-disk technology. Most textbooks on databases include discussions of the material presented here. Most data structures textbooks, including Knuth (1998), discuss static hashing in more detail; Knuth has

a complete discussion of hash functions and collision resolution techniques, as well as of their performance comparison. Knuth also offers a detailed discussion of techniques for sorting external files. Textbooks on file structures include Claybrook (1992), Smith and Barnes (1987), and Salzberg (1988); they discuss additional file organizations including tree-structured files, and have detailed algorithms for operations on files. Salzberg et al. (1990) describe a distributed external sorting algorithm. File organizations with a high degree of fault tolerance are described by Bitton and Gray (1988) and by Gray et al. (1990). Disk striping was proposed in Salem and Garcia Molina (1986). The first paper on redundant arrays of inexpensive disks (RAID) is by Patterson et al. (1988). Chen and Patterson (1990) and the excellent survey of RAID by Chen et al. (1994) are additional references. Grochowski and Hoyt (1996) discuss future trends in disk drives. Various formulas for the RAID architecture appear in Chen et al. (1994).

Morris (1968) is an early paper on hashing. Extendible hashing is described in Fagin et al. (1979). Linear hashing is described by Litwin (1980). Algorithms for insertion and deletion for linear hashing are discussed with illustrations in Salzberg (1988). Dynamic hashing, which we briefly introduced, was proposed by Larson (1978). There are many proposed variations for extendible and linear hashing; for examples, see Cesarini and Soda (1991), Du and Tong (1991), and Hachem and Berra (1992).

Gibson et al. (1997) describe a file server scaling approach for network-attached storage, and Kubiatowicz et al. (2000) decribe the Oceanstore system for creating a global utility infrastructure for storing persistent data. Both are considered pioneering approaches that led to the ideas for object-based storage. Mesnier et al. (2003) give an overview of the object storage concept. The Lustre system (Braam & Schwan, 2002) was one of the first object storage products and is used in the majority of supercomputers, including the top two, namely China's Tianhe-2 and Oakridge National Lab's Titan.

Details of disk storage devices can be found at manufacturer sites (for example, http://www.seagate.com, http://www.ibm.com, http://www.emc.com, http://www.hp.com, http://www.storagetek.com). IBM has a storage technology research center at IBM Almaden (http://www.almaden.ibm.com). Additional useful sites include CISCO storage solutions at cisco.com; Network Appliance (NetApp) at www.netapp.com; Hitachi Data Storage (HDS) at www.hds.com, and SNIA (Storage Networking Industry Association) at www.snia.org. A number of industry white papers are available at the aforementioned sites.

This page intentionally left blank

Indexing Structures for Files and Physical Database Design

In this chapter, we assume that a file already exists with some primary organization such as the unordered, ordered, or hashed organizations that were described in Chapter 16. We will describe additional auxiliary **access structures** called **indexes**, which are used to speed up the retrieval of records in response to certain search conditions. The index structures are additional files on disk that provide **secondary access paths**, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk. They enable efficient access to records based on the **indexing fields** that are used to construct the index. Basically, *any field* of the file can be used to create an index, and *multiple indexes* on different fields—as well as indexes on *multiple fields*—can be constructed on the same file. A variety of indexes are possible; each of them uses a particular data structure to speed up the search. To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located. The most prevalent types of indexes are based on ordered files (single-level indexes) and use tree data structures (multilevel indexes, B^+-trees) to organize the index. Indexes can also be constructed based on hashing or other search data structures. We also discuss indexes that are vectors of bits called *bitmap indexes*.

We describe different types of single-level ordered indexes—primary, secondary, and clustering—in Section 17.1. By viewing a single-level index as an ordered file, one can develop additional indexes for it, giving rise to the concept of multilevel indexes. A popular indexing scheme called **ISAM (indexed sequential access method)** is based on this idea. We discuss multilevel tree-structured indexes in Section 17.2. In Section 17.3, we describe B-trees and B^+-trees, which are data structures that are commonly used in DBMSs to implement dynamically changing

601

multilevel indexes. B[+]-trees have become a commonly accepted default structure for generating indexes on demand in most relational DBMSs. Section 17.4 is devoted to alternative ways to access data based on a combination of multiple keys. In Section 17.5, we discuss hash indexes and introduce the concept of logical indexes, which give an additional level of indirection from physical indexes and allow the physical index to be flexible and extensible in its organization. In Section 17.6, we discuss multikey indexing and bitmap indexes used for searching on one or more keys. Section 17.7 covers physical design and Section 7.8 summarizes the chapter.

17.1 Types of Single-Level Ordered Indexes

The idea behind an ordered index is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search the book index for a certain term in the textbook to find a list of *addresses*—page numbers in this case—and use these addresses to locate the specified pages first and then *search* for the term on each specified page. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a *linear search*, which scans the whole file. Of course, most books do have additional information, such as chapter and section titles, which help us find a term without having to search through the whole book. However, the index is the only exact indication of the pages where each term occurs in the book.

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field** (or **indexing attribute**).[1] The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are *ordered* so that we can do a *binary search* on the index. If both the data file and the index file are ordered, and since the index file is typically much smaller than the data file, searching the index using a binary search is a better option. Tree-structured multilevel indexes (see Section 17.2) implement an extension of the binary search idea that reduces the search space by two-way partitioning at each search step to an *n*-ary partitioning approach that divides the search space in the file *n*-ways at each stage.

There are several types of ordered indexes. A **primary index** is specified on the *ordering key field* of an **ordered file** of records. Recall from Section 16.7 that an ordering key field is used to *physically order* the file records on disk, and every record has a *unique value* for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field—another type of index, called a **clustering index**, can be used. The data file is called a **clustered file** in this latter case. Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, *but*

---

[1]We use the terms *field* and *attribute* interchangeably in this chapter.

not both. A third type of index, called a **secondary index**, can be specified on any *nonordering* field of a file. A data file can have several secondary indexes in addition to its primary access method. We discuss these types of single-level indexes in the next three subsections.

17.1.1 Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file. The first field is of the same data type as the ordering key field—called the **primary key**—of the data file, and the second field is a pointer to a disk block (a block address). There is one **index entry** (or **index record**) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values. We will refer to the two field values of index entry *i* as <*K*(*i*), *P*(*i*)>. In the rest of this chapter, we refer to different types of index **entries** < *K* (*i*), *X* > as follows:

- *X* may be the physical address of a block (or page) in the file, as in the case of *P*(*i*) above.
- *X* may be the record address made up of a block address and a record id (or offset) within the block.
- *X* may be a logical address of the block or of the record within the file and is a relative number that would be mapped to a physical address (see further explanation in Section 17.6.1).

To create a primary index on the ordered file shown in Figure 16.7, we use the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique). Each entry in the index has a Name value and a pointer. The first three index entries are as follows:
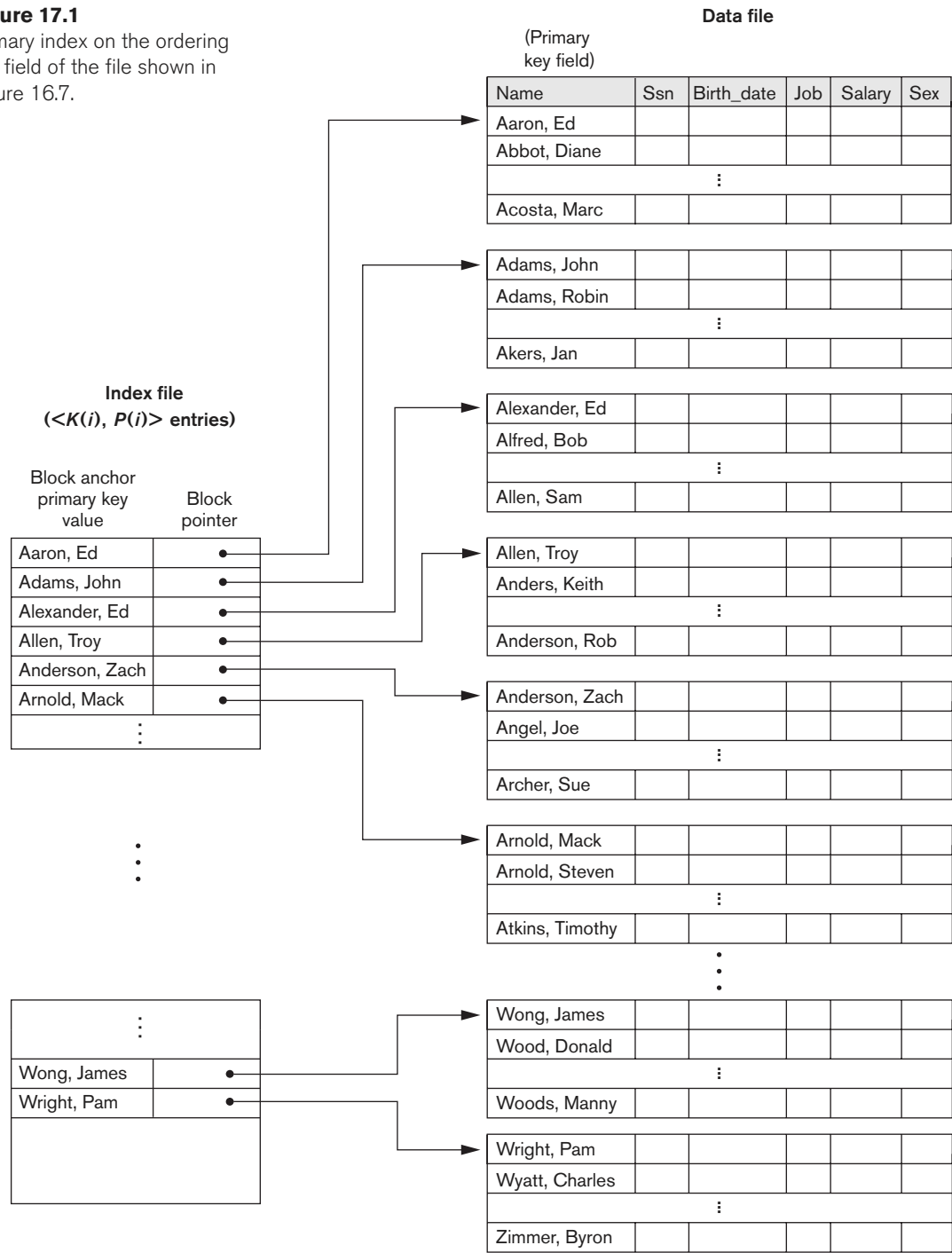
<*K*(1) = (Aaron, Ed), *P*(1) = address of block 1>
<*K*(2) = (Adams, John), *P*(2) = address of block 2>
<*K*(3) = (Alexander, Ed), *P*(3) = address of block 3>

Figure 17.1 illustrates this primary index. The total number of entries in the index is the same as the *number of disk blocks* in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.[2]

Indexes can also be characterized as dense or sparse. A **dense index** has an index entry for *every search key value* (and hence every record) in the data file. A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file. Thus, a primary index is a nondense (sparse) index, since it includes an

---

[2]We can use a scheme similar to the one described here, with the last record in each block (rather than the first) as the block anchor. This slightly improves the efficiency of the search algorithm.

Figure 17.1

Primary index on the ordering key field of the file shown in Figure 16.7.

entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).[3]

The index file for a primary index occupies a much smaller space than does the data file, for two reasons. First, there are *fewer index entries* than there are records in the data file. Second, each index entry is typically *smaller in size* than a data record because it has only two fields, both of which tend to be short in size; consequently, more index entries than data records can fit in one block. Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file. Referring to Table 16.3, note that the binary search for an ordered data file required $\log_2 b$ block accesses. But if the primary index file contains only b_i blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.

A record whose primary key value is K lies in the block whose address is $P(i)$, where $K(i) \leq K < K(i + 1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is $P(i)$.[4] Example 1 illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.

Example 1. Suppose that we have an ordered file with $r = 300,000$ records stored on a disk with block size $B = 4,096$ bytes.[5] File records are of fixed size and are unspanned, with record length $R = 100$ bytes. The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (4,096/100) \rfloor = 40$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (300,000/40) \rceil = 7,500$ blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 7,500) \rceil = 13$ block accesses.

Now suppose that the ordering key field of the file is $V = 9$ bytes long, a block pointer is $P = 6$ bytes long, and we have constructed a primary index for the file. The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/15) \rfloor = 273$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7,500/273) \rceil = 28$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses. To search for a record using the index, we need one additional block access to the data file for a total of $5 + 1 = 6$ block accesses—an improvement over binary search on the data file, which required 13 disk block accesses. Note that the index with 7,500 entries of 15 bytes each is rather small (112,500 or 112.5 Kbytes) and would typically be kept in main memory thus requiring negligible time to search with binary search. In that case we simply make one block access to retrieve the record.

---

[3]The sparse primary index has been called clustered (primary) index in some books and articles.

[4]Notice that the above formula would not be correct if the data file were ordered on a *nonkey field*; in that case the same index value in the block anchor could be repeated in the last records of the previous block.

[5]Most DBMS vendors, including Oracle, are using 4K or 4,096 bytes as a standard block/page size.