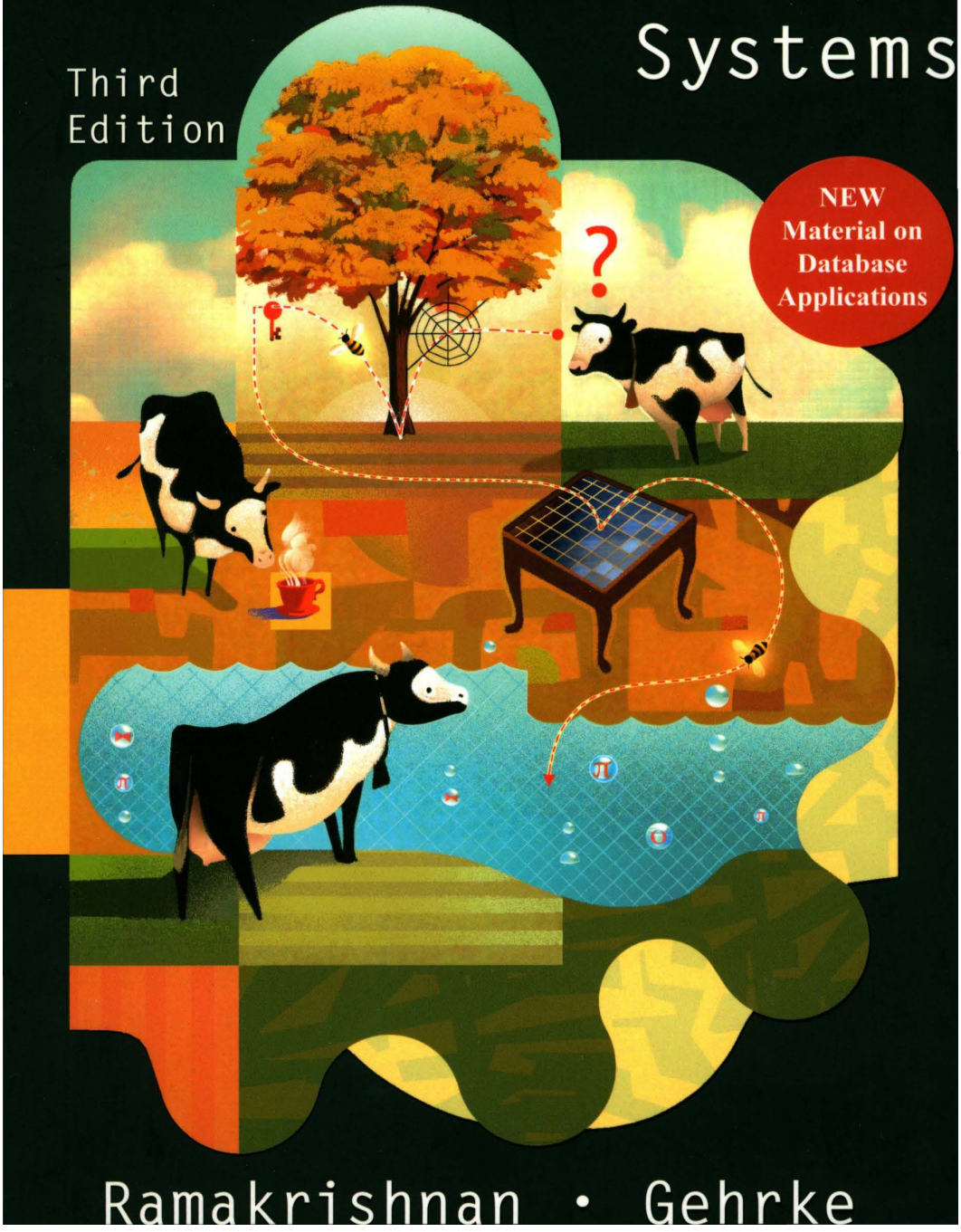
Database Management Systems

Third Edition



Ramakrishnan • Gehrke

DATABASE MANAGEMENT SYSTEMS

DATABASE MANAGEMENT SYSTEMS

Third Edition

Raghu Ramakrishnan
University of Wisconsin
Madison, Wisconsin, USA

Johannes Gehrke
Cornell University
Ithaca, New York, USA

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

McGraw-Hill Higher Education

A Division of The McGraw-Hill Companies

DATABASE MANAGEMENT SYSTEMS, THIRD EDITION

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

This book is printed on acid-free paper.

International 15 16 17 18 19 20 DOC/DOC 0 9 8 7 6 5
Domestic 8 9 0 DOC/DOC 0 9 8

ISBN-13: 978-0-07-246563-1
ISBN-10: 0-07-246563-8

ISBN-13: 978-0-07-115110-8 (ISE)
ISBN-10: 0-07-115110-9 (ISE)

Publisher: *Elizabeth A. Jones*
Senior developmental editor: *Emily J. Lupash*
Executive marketing manager: *John Wannemacher*
Senior project manager: *Gloria G. Schiesl*
Production supervisor: *Sherry L. Kane*
Media project manager: *Jodi K. Banowetz*
Senior media technology producer: *Phillip Meek*
Coordinator of freelance design: *Michelle D. Whitaker*
Cover illustration: *Mick Wiggins*
Compositor: *Interactive Composition Corporation*
Typeface: *11/13 Times Roman CMR 10*
Printer: *R. R. Donnelley/Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Ramakrishnan, Raghu.
Database management systems / Raghu Ramakrishnan, Johannes Gehrke. — 3rd ed.
p. cm.
Includes index.
ISBN 0–07–246563–8 — ISBN 0–07–115110–9 (ISE)
1. Database management. I. Gehrke, Johannes. II. Title.

QA76.9.D3 R237 2003
005.74—dc21 2002075205
CIP

INTERNATIONAL EDITION ISBN 0–07–115110–9

www.mhhe.com

To Apu, Ketan, and Vivek with love

To Keiko and Elisa

CONTENTS

PREFACE

> The advantage of doing one's praising for oneself is that one can lay it on so thick and exactly in the right places.
>
> —Samuel Butler

Database management systems are now an indispensable tool for managing information, and a course on the principles and practice of database systems is now an integral part of computer science curricula. This book covers the fundamentals of modern database management systems, in particular relational database systems.

We have attempted to present the material in a clear, simple style. A quantitative approach is used throughout with many detailed examples. An extensive set of exercises (for which solutions are available online to instructors) accompanies each chapter and reinforces students' ability to apply the concepts to real problems.

The book can be used with the accompanying software and programming assignments in two distinct kinds of introductory courses:

1. **Applications Emphasis:** A course that covers the principles of database systems, and emphasizes how they are used in developing data-intensive applications. Two new chapters on application development (one on database-backed applications, and one on Java and Internet application architectures) have been added to the third edition, and the entire book has been extensively revised and reorganized to support such a course. A running case-study and extensive online materials (e.g., code for SQL queries and Java applications, online databases and solutions) make it easy to teach a hands-on application-centric course.
2. **Systems Emphasis:** A course that has a strong systems emphasis and assumes that students have good programming skills in C and C++. In this case the accompanying Minibase software can be used as the basis for projects in which students are asked to implement various parts of a relational DBMS. Several central modules in the project software (e.g., heap files, buffer manager, B+ trees, hash indexes, various join methods)

are described in sufficient detail in the text to enable students to implement them, given the (C++) class interfaces.

Many instructors will no doubt teach a course that falls between these two extremes. The restructuring in the third edition offers a very modular organization that facilitates such hybrid courses. The also book contains enough material to support advanced courses in a two-course sequence.

Organization of the Third Edition

The book is organized into six main parts plus a collection of advanced topics, as shown in Figure 0.1. The Foundations chapters introduce database systems, the

(1) Foundations	Both
(2) Application Development	Applications emphasis
(3) Storage and Indexing	Systems emphasis
(4) Query Evaluation	Systems emphasis
(5) Transaction Management	Systems emphasis
(6) Database Design and Tuning	Applications emphasis
(7) Additional Topics	Both

Figure 0.1 Organization of Parts in the Third Edition

ER model and the relational model. They explain how databases are created and used, and cover the basics of database design and querying, including an in-depth treatment of SQL queries. While an instructor can omit some of this material at their discretion (e.g., relational calculus, some sections on the ER model or SQL queries), this material is relevant to every student of database systems, and we recommend that it be covered in as much detail as possible.

Each of the remaining five main parts has either an application or a systems emphasis. Each of the three Systems parts has an overview chapter, designed to provide a self-contained treatment, e.g., Chapter 8 is an overview of storage and indexing. The overview chapters can be used to provide stand-alone coverage of the topic, or as the first chapter in a more detailed treatment. Thus, in an application-oriented course, Chapter 8 might be the only material covered on file organizations and indexing, whereas in a systems-oriented course it would be supplemented by a selection from Chapters 9 through 11. The Database Design and Tuning part contains a discussion of performance tuning and designing for secure access. These application topics are best covered after giving students a good grasp of database system architecture, and are therefore placed later in the chapter sequence.

Suggested Course Outlines

The book can be used in two kinds of introductory database courses, one with an applications emphasis and one with a systems emphasis.

The *introductory applications-oriented course* could cover the Foundations chapters, then the Application Development chapters, followed by the overview systems chapters, and conclude with the Database Design and Tuning material. Chapter dependencies have been kept to a minimum, enabling instructors to easily fine tune what material to include. The Foundations material, Part I, should be covered first, and within Parts III, IV, and V, the overview chapters should be covered first. The only remaining dependencies between chapters in Parts I to VI are shown as arrows in Figure 0.2. The chapters in Part I should be covered in sequence. However, the coverage of algebra and calculus can be skipped in order to get to SQL queries sooner (although we believe this material is important and recommend that it should be covered before SQL).

The *introductory systems-oriented course* would cover the Foundations chapters and a selection of Applications and Systems chapters. An important point for systems-oriented courses is that the timing of programming projects (e.g., using Minibase) makes it desirable to cover some systems topics early. Chapter dependencies have been carefully limited to allow the Systems chapters to be covered as soon as Chapters 1 and 3 have been covered. The remaining Foundations chapters and Applications chapters can be covered subsequently.

The book also has ample material to support a multi-course sequence. Obviously, choosing an applications or systems emphasis in the introductory course results in dropping certain material from the course; the material in the book supports a comprehensive two-course sequence that covers both applications and systems aspects. The Additional Topics range over a broad set of issues, and can be used as the core material for an advanced course, supplemented with further readings.

Supplementary Material

This book comes with extensive online supplements:

- **Online Chapter:** To make space for new material such as application development, information retrieval, and XML, we've moved the coverage of QBE to an online chapter. Students can freely download the chapter from the book's web site, and solutions to exercises from this chapter are included in solutions manual.

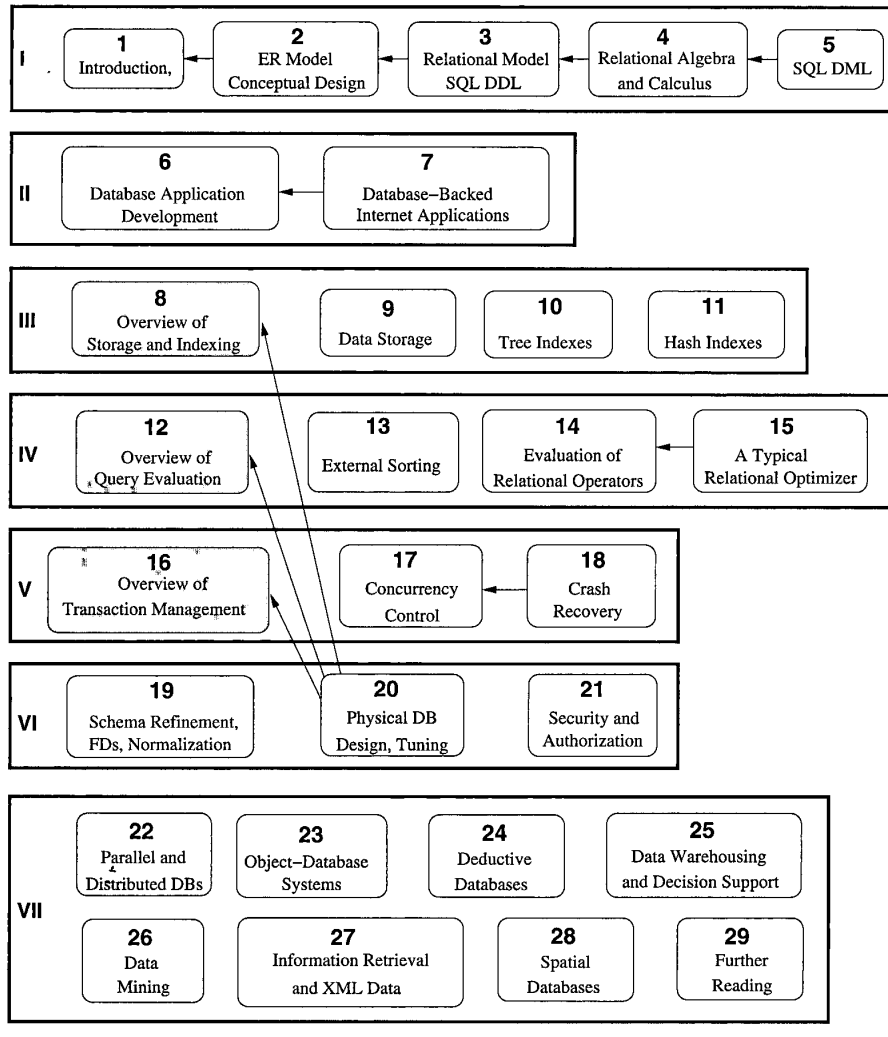


Figure 0.2 Chapter Organization and Dependencies

- **Lecture Slides:** Lecture slides are freely available for all chapters in Postscript, and PDF formats. Course instructors can also obtain these slides in Microsoft Powerpoint format, and can adapt them to their teaching needs. Instructors also have access to all figures used in the book (in xfig format), and can use them to modify the slides.

- **Solutions to Chapter Exercises:** The book has an unusually extensive set of in-depth exercises. Students can obtain solutions to odd-numbered chapter exercises and a set of lecture slides for each chapter through the Web in Postscript and Adobe PDF formats. Course instructors can obtain solutions to all exercises.
- **Software:** The book comes with two kinds of software. First, we have Minibase, a small relational DBMS intended for use in systems-oriented courses. Minibase comes with sample assignments and solutions, as described in Appendix 30. Access is restricted to course instructors. Second, we offer code for all SQL and Java application development exercises in the book, together with scripts to create sample databases, and scripts for setting up several commercial DBMSs. Students can only access solution code for odd-numbered exercises, whereas instructors have access to all solutions.
- **Instructor's Manual:** The book comes with an online manual that offers instructors comments on the material in each chapter. It provides a summary of each chapter and identifies choices for material to emphasize or omit. The manual also discusses the on-line supporting material for that chapter and offers numerous suggestions for hands-on exercises and projects. Finally, it includes samples of examination papers from courses taught by the authors using the book. It is restricted to course instructors.

For More Information

The home page for this book is at URL:

```
http://www.cs.wisc.edu/~dbbook
```

It contains a list of the changes between the 2nd and 3rd editions, and a frequently updated *link to all known errors in the book and its accompanying supplements.* Instructors should visit this site periodically or register at this site to be notified of important changes by email.

Acknowledgments

This book grew out of lecture notes for CS564, the introductory (senior/graduate level) database course at UW-Madison. David DeWitt developed this course and the Minirel project, in which students wrote several well-chosen parts of a relational DBMS. My thinking about this material was shaped by teaching CS564, and Minirel was the inspiration for Minibase, which is more comprehensive (e.g., it has a query optimizer and includes visualization software) but

tries to retain the spirit of Minirel. Mike Carey and I jointly designed much of Minibase. My lecture notes (and in turn this book) were influenced by Mike's lecture notes and by Yannis Ioannidis's lecture slides.

Joe Hellerstein used the beta edition of the book at Berkeley and provided invaluable feedback, assistance on slides, and hilarious quotes. Writing the chapter on object-database systems with Joe was a lot of fun.

C. Mohan provided invaluable assistance, patiently answering a number of questions about implementation techniques used in various commercial systems, in particular indexing, concurrency control, and recovery algorithms. Moshe Zloof answered numerous questions about QBE semantics and commercial systems based on QBE. Ron Fagin, Krishna Kulkarni, Len Shapiro, Jim Melton, Dennis Shasha, and Dirk Van Gucht reviewed the book and provided detailed feedback, greatly improving the content and presentation. Michael Goldweber at Beloit College, Matthew Haines at Wyoming, Michael Kifer at SUNY StonyBrook, Jeff Naughton at Wisconsin, Praveen Seshadri at Cornell, and Stan Zdonik at Brown also used the beta edition in their database courses and offered feedback and bug reports. In particular, Michael Kifer pointed out an error in the (old) algorithm for computing a minimal cover and suggested covering some SQL features in Chapter 2 to improve modularity. Gio Wiederhold's bibliography, converted to Latex format by S. Sudarshan, and Michael Ley's online bibliography on databases and logic programming were a great help while compiling the chapter bibliographies. Shaun Flisakowski and Uri Shaft helped me frequently in my never-ending battles with Latex.

I owe a special thanks to the many, many students who have contributed to the Minibase software. Emmanuel Ackaouy, Jim Pruyne, Lee Schumacher, and Michael Lee worked with me when I developed the first version of Minibase (much of which was subsequently discarded, but which influenced the next version). Emmanuel Ackaouy and Bryan So were my TAs when I taught CS564 using this version and went well beyond the limits of a TAship in their efforts to refine the project. Paul Aoki struggled with a version of Minibase and offered lots of useful comments as a TA at Berkeley. An entire class of CS764 students (our graduate database course) developed much of the current version of Minibase in a large class project that was led and coordinated by Mike Carey and me. Amit Shukla and Michael Lee were my TAs when I first taught CS564 using this version of Minibase and developed the software further.

Several students worked with me on independent projects, over a long period of time, to develop Minibase components. These include visualization packages for the buffer manager and B+ trees (Huseyin Bektas, Harry Stavropoulos, and Weiqing Huang); a query optimizer and visualizer (Stephen Harris, Michael Lee, and Donko Donjerkovic); an ER diagram tool based on the Opossum schema

editor (Eben Haber); and a GUI-based tool for normalization (Andrew Prock and Andy Therber). In addition, Bill Kimmel worked to integrate and fix a large body of code (storage manager, buffer manager, files and access methods, relational operators, and the query plan executor) produced by the CS764 class project. Ranjani Ramamurty considerably extended Bill's work on cleaning up and integrating the various modules. Luke Blanshard, Uri Shaft, and Shaun Flisakowski worked on putting together the release version of the code and developed test suites and exercises based on the Minibase software. Krishna Kunchithapadam tested the optimizer and developed part of the Minibase GUI.

Clearly, the Minibase software would not exist without the contributions of a great many talented people. With this software available freely in the public domain, I hope that more instructors will be able to teach a systems-oriented database course with a blend of implementation and experimentation to complement the lecture material.

I'd like to thank the many students who helped in developing and checking the solutions to the exercises and provided useful feedback on draft versions of the book. In alphabetical order: X. Bao, S. Biao, M. Chakrabarti, C. Chan, W. Chen, N. Cheung, D. Colwell, C. Fritz, V. Ganti, J. Gehrke, G. Glass, V. Gopalakrishnan, M. Higgins, T. Jasmin, M. Krishnaprasad, Y. Lin, C. Liu, M. Lusignan, H. Modi, S. Narayanan, D. Randolph, A. Ranganathan, J. Reminga, A. Therber, M. Thomas, Q. Wang, R. Wang, Z. Wang, and J. Yuan. Arcady Grenader, James Harrington, and Martin Reames at Wisconsin and Nina Tang at Berkeley provided especially detailed feedback.

Charlie Fischer, Avi Silberschatz, and Jeff Ullman gave me invaluable advice on working with a publisher. My editors at McGraw-Hill, Betsy Jones and Eric Munson, obtained extensive reviews and guided this book in its early stages. Emily Gray and Brad Kosirog were there whenever problems cropped up. At Wisconsin, Ginny Werner really helped me to stay on top of things.

Finally, this book was a thief of time, and in many ways it was harder on my family than on me. My sons expressed themselves forthrightly. From my (then) five-year-old, Ketan: "Dad, stop working on that silly book. You don't have any time for *me*." Two-year-old Vivek: "You working *boook*? No no no come play basketball me!" All the seasons of their discontent were visited upon my wife, and Apu nonetheless cheerfully kept the family going in its usual chaotic, happy way all the many evenings and weekends I was wrapped up in this book. (Not to mention the days when I was wrapped up in being a faculty member!) As in all things, I can trace my parents' hand in much of this; my father, with his love of learning, and my mother, with her love of us, shaped me. My brother Kartik's contributions to this book consisted chiefly of phone calls in which he kept me from working, but if I don't acknowledge him, he's liable to

be annoyed. I'd like to thank my family for being there and giving meaning to everything I do. (There! I knew I'd find a legitimate reason to thank Kartik.)

Acknowledgments for the Second Edition

Emily Gray and Betsy Jones at McGraw-Hill obtained extensive reviews and provided guidance and support as we prepared the second edition. Jonathan Goldstein helped with the bibliography for spatial databases. The following reviewers provided valuable feedback on content and organization: Liming Cai at Ohio University, Costas Tsatsoulis at University of Kansas, Kwok-Bun Yue at University of Houston, Clear Lake, William Grosky at Wayne State University, Sang H. Son at University of Virginia, James M. Slack at Minnesota State University, Mankato, Herman Balsters at University of Twente, Netherlands, Karen C. Davis at University of Cincinnati, Joachim Hammer at University of Florida, Fred Petry at Tulane University, Gregory Speegle at Baylor University, Salih Yurttas at Texas A&M University, and David Chao at San Francisco State University.

A number of people reported bugs in the first edition. In particular, we wish to thank the following: Joseph Albert at Portland State University, Han-yin Chen at University of Wisconsin, Lois Delcambre at Oregon Graduate Institute, Maggie Eich at Southern Methodist University, Raj Gopalan at Curtin University of Technology, Davood Rafiei at University of Toronto, Michael Schrefl at University of South Australia, Alex Thomasian at University of Connecticut, and Scott Vandenberg at Siena College.

A special thanks to the many people who answered a detailed survey about how commercial systems support various features: At IBM, Mike Carey, Bruce Lindsay, C. Mohan, and James Teng; at Informix, M. Muralikrishna and Michael Ubell; at Microsoft, David Campbell, Goetz Graefe, and Peter Spiro; at Oracle, Hakan Jacobsson, Jonathan D. Klein, Muralidhar Krishnaprasad, and M. Ziauddin; and at Sybase, Marc Chanliau, Lucien Dimino, Sangeeta Doraiswamy, Hanuma Kodavalla, Roger MacNicol, and Tirumanjanam Rengarajan.

After reading about himself in the acknowledgment to the first edition, Ketan (now 8) had a simple question: "How come you didn't dedicate the book to us? Why mom?" Ketan, I took care of this inexplicable oversight. Vivek (now 5) was more concerned about the extent of his fame: "Daddy, is my name in *evvy* copy of your book? Do they have it in *evvy* compooter science department in the world?" Vivek, I hope so. Finally, this revision would not have made it without Apu's and Keiko's support.

Acknowledgments for the Third Edition

We thank Raghav Kaushik for his contribution to the discussion of XML, and Alex Thomasian for his contribution to the coverage of concurrency control. A special thanks to Jim Melton for giving us a pre-publication copy of his book on object-oriented extensions in the SQL:1999 standard, and catching several bugs in a draft of this edition. Marti Hearst at Berkeley generously permitted us to adapt some of her slides on Information Retrieval, and Alon Levy and Dan Suciu were kind enough to let us adapt some of their lectures on XML. Mike Carey offered input on Web services.

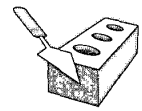
Emily Lupash at McGraw-Hill has been a source of constant support and encouragement. She coordinated extensive reviews from Ming Wang at Embry-Riddle Aeronautical University, Cheng Hsu at RPI, Paul Bergstein at Univ. of Massachusetts, Archana Sathaye at SJSU, Bharat Bhargava at Purdue, John Fendrich at Bradley, Ahmet Ugur at Central Michigan, Richard Osborne at Univ. of Colorado, Akira Kawaguchi at CCNY, Mark Last at Ben Gurion, Vassilis Tsotras at Univ. of California, and Ronald Eaglin at Univ. of Central Florida. It is a pleasure to acknowledge the thoughtful input we received from the reviewers, which greatly improved the design and content of this edition. Gloria Schiesl and Jade Moran dealt cheerfully and efficiently with last-minute snafus, and, with Sherry Kane, made a very tight schedule possible. Michelle Whitaker iterated many times on the cover and end-sheet design.

On a personal note for Raghu, Ketan, following the canny example of the camel that shared a tent, observed that "it is only fair" that Raghu dedicate this edition solely to him and Vivek, since "mommy already had it dedicated only to her." Despite this blatant attempt to hog the limelight, enthusiastically supported by Vivek and viewed with the indulgent affection of a doting father, this book is also dedicated to Apu, for being there through it all.

For Johannes, this revision would not have made it without Keiko's support and inspiration and the motivation from looking at Elisa's peacefully sleeping face.

PART I

FOUNDATIONS



1

OVERVIEW OF DATABASE SYSTEMS

- What is a DBMS, in particular, a relational DBMS?
- Why should we consider a DBMS to manage data?
- How is application data represented in a DBMS?
- How is data in a DBMS retrieved and manipulated?
- How does a DBMS support concurrent access and protect data during system failures?
- What are the main components of a DBMS?
- Who is involved with databases in real life?
- **Key concepts:** database management, data independence, database design, data model; relational databases and queries; schemas, levels of abstraction; transactions, concurrency and locking, recovery and logging; DBMS architecture; database administrator, application programmer, end user

Has everyone noticed that all the letters of the word *database* are typed with the left hand? Now the layout of the QWERTY typewriter keyboard was designed, among other things, to facilitate the even use of both hands. It follows, therefore, that writing about databases is not only unnatural, but a lot harder than it appears.

—Anonymous

The amount of information available to us is literally exploding, and the value of data as an organizational asset is widely recognized. To get the most out of their large and complex datasets, users require tools that simplify the tasks of

The area of database management systems is a microcosm of computer science in general. The issues addressed and the techniques used span a wide spectrum, including languages, object-orientation and other programming paradigms, compilation, operating systems, concurrent programming, data structures, algorithms, theory, parallel and distributed systems, user interfaces, expert systems and artificial intelligence, statistical techniques, and dynamic programming. We cannot go into all these aspects of database management in one book, but we hope to give the reader a sense of the excitement in this rich and vibrant discipline.

managing the data and extracting useful information in a timely fashion. Otherwise, data can become a liability, with the cost of acquiring it and managing it far exceeding the value derived from it.

A **database** is a collection of data, typically describing the activities of one or more related organizations. For example, a university database might contain information about the following:

- *Entities* such as students, faculty, courses, and classrooms.
- *Relationships* between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

A **database management system**, or **DBMS**, is software designed to assist in maintaining and utilizing large collections of data. The need for such systems, as well as their use, is growing rapidly. The alternative to using a DBMS is to store the data in files and write application-specific code to manage it. The use of a DBMS has several important advantages, as we will see in Section 1.4.

1.1 MANAGING DATA

The goal of this book is to present an in-depth introduction to database management systems, with an emphasis on how to *design* a database and *use* a DBMS effectively. Not surprisingly, many decisions about how to use a DBMS for a given application depend on what capabilities the DBMS supports efficiently. Therefore, to use a DBMS well, it is necessary to also understand how a DBMS *works*.

Many kinds of database management systems are in use, but this book concentrates on **relational database systems (RDBMSs)**, which are by far the dominant type of DBMS today. The following questions are addressed in the core chapters of this book:

1. **Database Design and Application Development:** How can a user describe a real-world enterprise (e.g., a university) in terms of the data stored in a DBMS? What factors must be considered in deciding how to organize the stored data? How can we develop applications that rely upon a DBMS? (Chapters 2, 3, 6, 7, 19, 20, and 21.)

2. **Data Analysis:** How can a user answer questions about the enterprise by posing queries over the data in the DBMS? (Chapters 4 and 5.)[1]

3. **Concurrency and Robustness:** How does a DBMS allow many users to access data concurrently, and how does it protect the data in the event of system failures? (Chapters 16, 17, and 18.)

4. **Efficiency and Scalability:** How does a DBMS store large datasets and answer questions against this data efficiently? (Chapters 8, 9, 10, 11, 12, 13, 14, and 15.)

Later chapters cover important and rapidly evolving topics, such as parallel and distributed database management, data warehousing and complex queries for decision support, data mining, databases and information retrieval, XML repositories, object databases, spatial data management, and rule-oriented DBMS extensions.

In the rest of this chapter, we introduce these issues. In Section 1.2, we begin with a brief history of the field and a discussion of the role of database management in modern information systems. We then identify the benefits of storing data in a DBMS instead of a file system in Section 1.3, and discuss the advantages of using a DBMS to manage data in Section 1.4. In Section 1.5, we consider how information about an enterprise should be organized and stored in a DBMS. A user probably thinks about this information in high-level terms that correspond to the entities in the organization and their relationships, whereas the DBMS ultimately stores data in the form of (many, many) bits. The gap between how users think of their data and how the data is ultimately stored is bridged through several *levels of abstraction* supported by the DBMS. Intuitively, a user can begin by describing the data in fairly high-level terms, then refine this description by considering additional storage and representation details as needed.

In Section 1.6, we consider how users can retrieve data stored in a DBMS and the need for techniques to efficiently compute answers to questions involving such data. In Section 1.7, we provide an overview of how a DBMS supports concurrent access to data by several users and how it protects the data in the event of system failures.

[1] An online chapter on Query-by-Example (QBE) is also available.

We then briefly describe the internal structure of a DBMS in Section 1.8, and mention various groups of people associated with the development and use of a DBMS in Section 1.9.

1.2 A HISTORICAL PERSPECTIVE

From the earliest days of computers, storing and manipulating data have been a major application focus. The first general-purpose DBMS, designed by Charles Bachman at General Electric in the early 1960s, was called the Integrated Data Store. It formed the basis for the *network data model*, which was standardized by the Conference on Data Systems Languages (CODASYL) and strongly influenced database systems through the 1960s. Bachman was the first recipient of ACM's Turing Award (the computer science equivalent of a Nobel Prize) for work in the database area; he received the award in 1973.

In the late 1960s, IBM developed the Information Management System (IMS) DBMS, used even today in many major installations. IMS formed the basis for an alternative data representation framework called the *hierarchical data model*. The SABRE system for making airline reservations was jointly developed by American Airlines and IBM around the same time, and it allowed several people to access the same data through a computer network. Interestingly, today the same SABRE system is used to power popular Web-based travel services such as Travelocity.

In 1970, Edgar Codd, at IBM's San Jose Research Laboratory, proposed a new data representation framework called the *relational data model*. This proved to be a watershed in the development of database systems: It sparked the rapid development of several DBMSs based on the relational model, along with a rich body of theoretical results that placed the field on a firm foundation. Codd won the 1981 Turing Award for his seminal work. Database systems matured as an academic discipline, and the popularity of relational DBMSs changed the commercial landscape. Their benefits were widely recognized, and the use of DBMSs for managing corporate data became standard practice.

In the 1980s, the relational model consolidated its position as the dominant DBMS paradigm, and database systems continued to gain widespread use. The SQL query language for relational databases, developed as part of IBM's System R project, is now the standard query language. SQL was standardized in the late 1980s, and the current standard, SQL:1999, was adopted by the American National Standards Institute (ANSI) and International Organization for Standardization (ISO). Arguably, the most widely used form of concurrent programming is the concurrent execution of database programs (called *transactions*). Users write programs as if they are to be run by themselves, and

the responsibility for running them concurrently is given to the DBMS. James Gray won the 1999 Turing award for his contributions to database transaction management.

In the late 1980s and the 1990s, advances were made in many areas of database systems. Considerable research was carried out into more powerful query languages and richer data models, with emphasis placed on supporting complex analysis of data from all parts of an enterprise. Several vendors (e.g., IBM's DB2, Oracle 8, Informix[2] UDS) extended their systems with the ability to store new data types such as images and text, and to ask more complex queries. Specialized systems have been developed by numerous vendors for creating *data warehouses*, consolidating data from several databases, and for carrying out specialized analysis.

An interesting phenomenon is the emergence of several **enterprise resource planning (ERP)** and **management resource planning (MRP)** packages, which add a substantial layer of application-oriented features on top of a DBMS. Widely used packages include systems from Baan, Oracle, PeopleSoft, SAP, and Siebel. These packages identify a set of common tasks (e.g., inventory management, human resources planning, financial analysis) encountered by a large number of organizations and provide a general application layer to carry out these tasks. The data is stored in a relational DBMS and the application layer can be customized to different companies, leading to lower overall costs for the companies, compared to the cost of building the application layer from scratch.

Most significant, perhaps, DBMSs have entered the Internet Age. While the first generation of websites stored their data exclusively in operating systems files, the use of a DBMS to store data accessed through a Web browser is becoming widespread. Queries are generated through Web-accessible forms and answers are formatted using a markup language such as HTML to be easily displayed in a browser. All the database vendors are adding features to their DBMS aimed at making it more suitable for deployment over the Internet.

Database management continues to gain importance as more and more data is brought online and made ever more accessible through computer networking. Today the field is being driven by exciting visions such as multimedia databases, interactive video, streaming data, digital libraries, a host of scientific projects such as the human genome mapping effort and NASA's Earth Observation System project, and the desire of companies to consolidate their decision-making processes and *mine* their data repositories for useful information about their businesses. Commercially, database management systems represent one of the

[2]Informix was recently acquired by IBM.

largest and most vigorous market segments. Thus the study of database systems could prove to be richly rewarding in more ways than one!

1.3 FILE SYSTEMS VERSUS A DBMS

To understand the need for a DBMS, let us consider a motivating scenario: A company has a large collection (say, 500 GB[3]) of data on employees, departments, products, sales, and so on. This data is accessed concurrently by several employees. Questions about the data must be answered quickly, changes made to the data by different users must be applied consistently, and access to certain parts of the data (e.g., salaries) must be restricted.

We can try to manage the data by storing it in operating system files. This approach has many drawbacks, including the following:

- We probably do not have 500 GB of main memory to hold all the data. We must therefore store data in a storage device such as a disk or tape and bring relevant parts into main memory for processing as needed.
- Even if we have 500 GB of main memory, on computer systems with 32-bit addressing, we cannot refer directly to more than about 4 GB of data. We have to program some method of identifying all data items.
- We have to write special programs to answer each question a user may want to ask about the data. These programs are likely to be complex because of the large volume of data to be searched.
- We must protect the data from inconsistent changes made by different users accessing the data concurrently. If applications must address the details of such concurrent access, this adds greatly to their complexity.
- We must ensure that data is restored to a consistent state if the system crashes while changes are being made.
- Operating systems provide only a password mechanism for security. This is not sufficiently flexible to enforce security policies in which different users have permission to access different subsets of the data.

A DBMS is a piece of software designed to make the preceding tasks easier. By storing data in a DBMS rather than as a collection of operating system files, we can use the DBMS's features to manage the data in a robust and efficient manner. As the volume of data and the number of users grow—hundreds of gigabytes of data and thousands of users are common in current corporate databases—DBMS support becomes indispensable.

[3]A kilobyte (KB) is 1024 bytes, a megabyte (MB) is 1024 KBs, a gigabyte (GB) is 1024 MBs, a terabyte (TB) is 1024 GBs, and a petabyte (PB) is 1024 terabytes.

1.4 ADVANTAGES OF A DBMS

Using a DBMS to manage data has many advantages:

- **Data Independence:** Application programs should not, ideally, be exposed to details of data representation and storage. The DBMS provides an abstract view of the data that hides such details.
- **Efficient Data Access:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **Data Integrity and Security:** If data is always accessed through the DBMS, the DBMS can enforce integrity constraints. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, it can enforce *access controls* that govern what data is visible to different classes of users.
- **Data Administration:** When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.
- **Concurrent Access and Crash Recovery:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **Reduced Application Development Time:** Clearly, the DBMS supports important functions that are common to many applications accessing data in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick application development. DBMS applications are also likely to be more robust than similar stand-alone applications because many important tasks are handled by the DBMS (and do not have to be debugged and tested in the application).

Given all these advantages, is there ever a reason *not* to use a DBMS? Sometimes, yes. A DBMS is a complex piece of software, optimized for certain kinds of workloads (e.g., answering complex queries or handling many concurrent requests), and its performance may not be adequate for certain specialized applications. Examples include applications with tight real-time constraints or just a few well-defined critical operations for which efficient custom code must be written. Another reason for not using a DBMS is that an application may need to manipulate the data in ways not supported by the query language. In

such a situation, the abstract view of the data presented by the DBMS does not match the application's needs and actually gets in the way. As an example, relational databases do not support flexible analysis of text data (although vendors are now extending their products in this direction).

If specialized performance or data manipulation requirements are central to an application, the application may choose not to use a DBMS, especially if the added benefits of a DBMS (e.g., flexible querying, security, concurrent access, and crash recovery) are not required. In most situations calling for large-scale data management, however, DBMSs have become an indispensable tool.

1.5 DESCRIBING AND STORING DATA IN A DBMS

The user of a DBMS is ultimately concerned with some real-world enterprise, and the data to be stored describes various aspects of this enterprise. For example, there are students, faculty, and courses in a university, and the data in a university database describes these entities and their relationships.

A **data model** is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the **relational data model**, which we focus on in this book.

While the data model of the DBMS hides many details, it is nonetheless closer to how the DBMS stores data than to how a user thinks about the underlying enterprise. A **semantic data model** is a more abstract, high-level data model that makes it easier for a user to come up with a good initial description of the data in an enterprise. These models contain a wide variety of constructs that help describe a real application scenario. A DBMS is not intended to support all these constructs directly; it is typically built around a data model with just a few basic constructs, such as the relational model. A database design in terms of a semantic model serves as a useful starting point and is subsequently translated into a database design in terms of the data model the DBMS actually supports.

A widely used semantic data model called the entity-relationship (ER) model allows us to pictorially denote entities and the relationships among them. We cover the ER model in Chapter 2.

An Example of Poor Design: The relational schema for Students illustrates a poor design choice; you should *never* create a field such as *age*, whose value is constantly changing. A better choice would be *DOB* (for *date of birth*); age can be computed from this. We continue to use *age* in our examples, however, because it makes them easier to read.

1.5.1 The Relational Model

In this section we provide a brief introduction to the relational model. The central data description construct in this model is a **relation**, which can be thought of as a set of **records**.

A description of data in terms of a data model is called a **schema**. In the relational model, the schema for a relation specifies its name, the name of each **field** (or **attribute** or **column**), and the type of each field. As an example, student information in a university database may be stored in a relation with the following schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)

The preceding schema says that each record in the Students relation has five fields, with field names and types as indicated. An example instance of the Students relation appears in Figure 1.1.

sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 1.1 An Instance of the Students Relation

Each row in the Students relation is a record that describes a student. The description is not complete—for example, the student's height is not included—but is presumably adequate for the intended applications in the university database. Every row follows the schema of the Students relation. The schema can therefore be regarded as a template for describing a student.

We can make the description of a collection of students more precise by specifying **integrity constraints**, which are conditions that the records in a relation

must satisfy. For example, we could specify that every student has a unique *sid* value. Observe that we cannot capture this information by simply adding another field to the Students schema. Thus, the ability to specify uniqueness of the values in a field increases the accuracy with which we can describe our data. The expressiveness of the constructs available for specifying integrity constraints is an important aspect of a data model.

Other Data Models

In addition to the relational data model (which is used in numerous systems, including IBM's DB2, Informix, Oracle, Sybase, Microsoft's Access, FoxBase, Paradox, Tandem, and Teradata), other important data models include the hierarchical model (e.g., used in IBM's IMS DBMS), the network model (e.g., used in IDS and IDMS), the object-oriented model (e.g., used in Objectstore and Versant), and the object-relational model (e.g., used in DBMS products from IBM, Informix, ObjectStore, Oracle, Versant, and others). While many databases use the hierarchical and network models and systems based on the object-oriented and object-relational models are gaining acceptance in the marketplace, the dominant model today is the relational model.

In this book, we focus on the relational model because of its wide use and importance. Indeed, the object-relational model, which is gaining in popularity, is an effort to combine the best features of the relational and object-oriented models, and a good grasp of the relational model is necessary to understand object-relational concepts. (We discuss the object-oriented and object-relational models in Chapter 23.)

1.5.2 Levels of Abstraction in a DBMS

The data in a DBMS is described at three levels of abstraction, as illustrated in Figure 1.2. The database description consists of a schema at each of these three levels of abstraction: the *conceptual*, *physical*, and *external*.

A **data definition language** (DDL) is used to define the external and conceptual schemas. We discuss the DDL facilities of the most widely used database language, SQL, in Chapter 3. All DBMS vendors also support SQL commands to describe aspects of the physical schema, but these commands are not part of the SQL language standard. Information about the conceptual, external, and physical schemas is stored in the **system catalogs** (Section 12.1). We discuss the three levels of abstraction in the rest of this section.

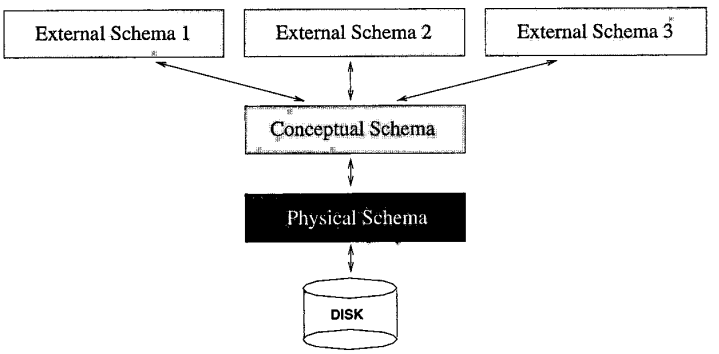


Figure 1.2 Levels of Abstraction in a DBMS

Conceptual Schema

The **conceptual schema** (sometimes called the **logical schema**) describes the stored data in terms of the data model of the DBMS. In a relational DBMS, the conceptual schema describes all relations that are stored in the database. In our sample university database, these relations contain information about *entities*, such as students and faculty, and about *relationships*, such as students' enrollment in courses. All student entities can be described using records in a Students relation, as we saw earlier. In fact, each collection of entities and each collection of relationships can be described as a relation, leading to the following conceptual schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)
Faculty(*fid:* `string`, *fname:* `string`, *sal:* `real`)
Courses(*cid:* `string`, *cname:* `string`, *credits:* `integer`)
Rooms(*rno:* `integer`, *address:* `string`, *capacity:* `integer`)
Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`)
Teaches(*fid:* `string`, *cid:* `string`)
Meets_In(*cid:* `string`, *rno:* `integer`, *time:* `string`)

The choice of relations, and the choice of fields for each relation, is not always obvious, and the process of arriving at a good conceptual schema is called **conceptual database design**. We discuss conceptual database design in Chapters 2 and 19.

Physical Schema

The **physical schema** specifies additional storage details. Essentially, the physical schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes.

We must decide what file organizations to use to store the relations and create auxiliary data structures, called **indexes**, to speed up data retrieval operations. A sample physical schema for the university database follows:

- Store all relations as unsorted files of records. (A file in a DBMS is either a collection of records or a collection of pages, rather than a string of characters as in an operating system.)
- Create indexes on the first column of the Students, Faculty, and Courses relations, the *sal* column of Faculty, and the *capacity* column of Rooms.

Decisions about the physical schema are based on an understanding of how the data is typically accessed. The process of arriving at a good physical schema is called **physical database design**. We discuss physical database design in Chapter 20.

External Schema

External schemas, which usually are also in terms of the data model of the DBMS, allow data access to be customized (and authorized) at the level of individual users or groups of users. Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users. Each external schema consists of a collection of one or more **views** and relations from the conceptual schema. A view is conceptually a relation, but the records in a view are not stored in the DBMS. Rather, they are computed using a definition for the view, in terms of relations stored in the DBMS. We discuss views in more detail in Chapters 3 and 25.

The external schema design is guided by end user requirements. For example, we might want to allow students to find out the names of faculty members teaching courses as well as course enrollments. This can be done by defining the following view:

Courseinfo(*cid:* `string`, *fname:* `string`, *enrollment:* `integer`)

A user can treat a view just like a relation and ask questions about the records in the view. Even though the records in the view are not stored explicitly,

they are computed as needed. We did not include Courseinfo in the conceptual schema because we can compute Courseinfo from the relations in the conceptual schema, and to store it in addition would be redundant. Such redundancy, in addition to the wasted space, could lead to inconsistencies. For example, a tuple may be inserted into the Enrolled relation, indicating that a particular student has enrolled in some course, without incrementing the value in the *enrollment* field of the corresponding record of Courseinfo (if the latter also is part of the conceptual schema and its tuples are stored in the DBMS).

1.5.3 Data Independence

A very important advantage of using a DBMS is that it offers **data independence**. That is, application programs are insulated from changes in the way the data is structured and stored. Data independence is achieved through use of the three levels of data abstraction; in particular, the conceptual schema and the external schema provide distinct benefits in this area.

Relations in the external schema (view relations) are in principle generated on demand from the relations corresponding to the conceptual schema.[4] If the underlying data is reorganized, that is, the conceptual schema is changed, the definition of a view relation can be modified so that the same relation is computed as before. For example, suppose that the Faculty relation in our university database is replaced by the following two relations:

Faculty_public(*fid:* `string`, *fname:* `string`, *office:* `integer`)
Faculty_private(*fid:* `string`, *sal:* `real`)

Intuitively, some confidential information about faculty has been placed in a separate relation and information about offices has been added. The Courseinfo view relation can be redefined in terms of Faculty_public and Faculty_private, which together contain all the information in Faculty, so that a user who queries Courseinfo will get the same answers as before.

Thus, users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This property is called **logical data independence**.

In turn, the conceptual schema insulates users from changes in physical storage details. This property is referred to as **physical data independence**. The conceptual schema hides details such as how the data is actually laid out on disk, the file structure, and the choice of indexes. As long as the conceptual

[4] In practice, they could be precomputed and stored to speed up queries on view relations, but the computed view relations must be updated whenever the underlying relations are updated.

schema remains the same, we can change these storage details without altering applications. (Of course, performance might be affected by such changes.)

1.6 QUERIES IN A DBMS

The ease with which information can be obtained from a database often determines its value to a user. In contrast to older database systems, relational database systems allow a rich class of questions to be posed easily; this feature has contributed greatly to their popularity. Consider the sample university database in Section 1.5.2. Here are some questions a user might ask:

1. What is the name of the student with student ID 123456?
2. What is the average salary of professors who teach course CS564?
3. How many students are enrolled in CS564?
4. What fraction of students in CS564 received a grade better than B?
5. Is any student with a GPA less than 3.0 enrolled in CS564?

Such questions involving the data stored in a DBMS are called **queries**. A DBMS provides a specialized language, called the **query language**, in which queries can be posed. A very attractive feature of the relational model is that it supports powerful query languages. **Relational calculus** is a formal query language based on mathematical logic, and queries in this language have an intuitive, precise meaning. **Relational algebra** is another formal query language, based on a collection of **operators** for manipulating relations, which is equivalent in power to the calculus.

A DBMS takes great care to evaluate queries as efficiently as possible. We discuss query optimization and evaluation in Chapters 12, 14, and 15. Of course, the efficiency of query evaluation is determined to a large extent by how the data is stored physically. Indexes can be used to speed up many queries—in fact, a good choice of indexes for the underlying relations can speed up each query in the preceding list. We discuss data storage and indexing in Chapters 8, 9, 10, and 11.

A DBMS enables users to create, modify, and query data through a **data manipulation language** (DML). Thus, the query language is only one part of the DML, which also provides constructs to insert, delete, and modify data. We will discuss the DML features of SQL in Chapter 5. The DML and DDL are collectively referred to as the **data sublanguage** when embedded within a **host language** (e.g., C or COBOL).

1.7 TRANSACTION MANAGEMENT

Consider a database that holds information about airline reservations. At any given instant, it is possible (and likely) that several travel agents are looking up information about available seats on various flights and making new seat reservations. When several users access (and possibly modify) a database concurrently, the DBMS must order their requests carefully to avoid conflicts. For example, when one travel agent looks up Flight 100 on some given day and finds an empty seat, another travel agent may simultaneously be making a reservation for that seat, thereby making the information seen by the first agent obsolete.

Another example of concurrent use is a bank's database. While one user's application program is computing the total deposits, another application may transfer money from an account that the first application has just 'seen' to an account that has not yet been seen, thereby causing the total to appear larger than it should be. Clearly, such anomalies should not be allowed to occur. However, disallowing concurrent access can degrade performance.

Further, the DBMS must protect users from the effects of system failures by ensuring that all data (and the status of active applications) is restored to a consistent state when the system is restarted after a crash. For example, if a travel agent asks for a reservation to be made, and the DBMS responds saying that the reservation has been made, the reservation should not be lost if the system crashes. On the other hand, if the DBMS has not yet responded to the request, but is making the necessary changes to the data when the crash occurs, the partial changes should be undone when the system comes back up.

A **transaction** is *any one execution* of a user program in a DBMS. (Executing the same program several times will generate several transactions.) This is the basic unit of change as seen by the DBMS: Partial transactions are not allowed, and the effect of a group of transactions is equivalent to some serial execution of all transactions. We briefly outline how these properties are guaranteed, deferring a detailed discussion to later chapters.

1.7.1 Concurrent Execution of Transactions

An important task of a DBMS is to schedule concurrent accesses to data so that each user can safely ignore the fact that others are accessing the data concurrently. The importance of this task cannot be underestimated because a database is typically shared by a large number of users, who submit their requests to the DBMS independently and simply cannot be expected to deal with arbitrary changes being made concurrently by other users. A DBMS

allows users to think of their programs as if they were executing in isolation, one after the other in some order chosen by the DBMS. For example, if a program that deposits cash into an account is submitted to the DBMS at the same time as another program that debits money from the same account, either of these programs could be run first by the DBMS, but their steps will not be interleaved in such a way that they interfere with each other.

A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. A **lock** is a mechanism used to control access to database objects. Two kinds of **locks** are commonly supported by a DBMS: **shared locks** on an object can be held by two different transactions at the same time, but an **exclusive lock** on an object ensures that no other transactions hold *any* lock on this object.

Suppose that the following locking protocol is followed: *Every transaction begins by obtaining a shared lock on each data object that it needs to read and an exclusive lock on each data object that it needs to modify, then releases all its locks after completing all actions.* Consider two transactions $T1$ and $T2$ such that $T1$ wants to modify a data object and $T2$ wants to read the same object. Intuitively, if $T1$'s request for an exclusive lock on the object is granted first, $T2$ cannot proceed until $T1$ releases this lock, because $T2$'s request for a shared lock will not be granted by the DBMS until then. Thus, all of $T1$'s actions will be completed before any of $T2$'s actions are initiated. We consider locking in more detail in Chapters 16 and 17.

1.7.2 Incomplete Transactions and System Crashes

Transactions can be interrupted before running to completion for a variety of reasons, e.g., a system crash. A DBMS must ensure that the changes made by such incomplete transactions are removed from the database. For example, if the DBMS is in the middle of transferring money from account A to account B and has debited the first account but not yet credited the second when the crash occurs, the money debited from account A must be restored when the system comes back up after the crash.

To do so, the DBMS maintains a **log** of all writes to the database. A crucial property of the log is that each write action must be recorded in the log (on disk) *before* the corresponding change is reflected in the database itself—otherwise, if the system crashes just after making the change in the database but before the change is recorded in the log, the DBMS would be unable to detect and undo this change. This property is called **Write-Ahead Log**, or **WAL**. To ensure

this property, the DBMS must be able to selectively force a page in memory to disk.

The log is also used to ensure that the changes made by a successfully completed transaction are not lost due to a system crash, as explained in Chapter 18. Bringing the database to a consistent state after a system crash can be a slow process, since the DBMS must ensure that the effects of all transactions that completed prior to the crash are restored, and that the effects of incomplete transactions are undone. The time required to recover from a crash can be reduced by periodically forcing some information to disk; this periodic operation is called a **checkpoint**.

1.7.3 Points to Note

In summary, there are three points to remember with respect to DBMS support for concurrency control and recovery:

1. Every object that is read or written by a transaction is first locked in shared or exclusive mode, respectively. Placing a lock on an object restricts its availability to other transactions and thereby affects performance.
2. For efficient log maintenance, the DBMS must be able to selectively force a collection of pages in main memory to disk. Operating system support for this operation is not always satisfactory.
3. Periodic checkpointing can reduce the time needed to recover from a crash. Of course, this must be balanced against the fact that checkpointing too often slows down normal execution.

1.8 STRUCTURE OF A DBMS

Figure 1.3 shows the structure (with some simplification) of a typical DBMS based on the relational data model.

The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a simplification: SQL commands can be embedded in host-language application programs, e.g., Java or COBOL programs. We ignore these issues to concentrate on the core DBMS functionality.)

When a user issues a query, the parsed query is presented to a **query optimizer**, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An **execution plan** is a

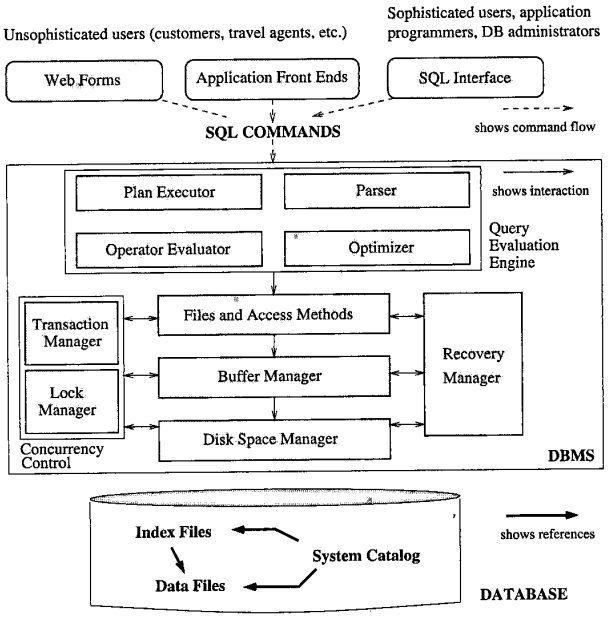


Figure 1.3 Architecture of a DBMS

blueprint for evaluating a query, usually represented as a tree of relational operators (with annotations that contain additional detailed information about which access methods to use, etc.). We discuss query optimization in Chapters 12 and 15. Relational operators serve as the building blocks for evaluating queries posed against the data. The implementation of these operators is discussed in Chapters 12 and 14.

The code that implements relational operators sits on top of the file and access methods layer. This layer supports the concept of a **file**, which, in a DBMS, is a collection of pages or a collection of records. **Heap files**, or files of unordered pages, as well as indexes are supported. In addition to keeping track of the pages in a file, this layer organizes the information within a page. File and page level storage issues are considered in Chapter 9. File organizations and indexes are considered in Chapter 8.

The files and access methods layer code sits on top of the **buffer manager**, which brings pages in from disk to main memory as needed in response to read requests. Buffer management is discussed in Chapter 9.

The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through (routines provided by) this layer, called the **disk space manager**. This layer is discussed in Chapter 9.

The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the **transaction manager**, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the **lock manager**, which keeps track of requests for locks and grants locks on database objects when they become available; and the **recovery manager**, which is responsible for maintaining a log and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and file and access method layers must interact with these components. We discuss concurrency control and recovery in detail in Chapter 16.

1.9 PEOPLE WHO WORK WITH DATABASES

Quite a variety of people are associated with the creation and use of databases. Obviously, there are **database implementors**, who build DBMS software, and **end users** who wish to store and use data in a DBMS. Database implementors work for vendors such as IBM or Oracle. End users come from a diverse and increasing number of fields. As data grows in complexity and volume, and is increasingly recognized as a major asset, the importance of maintaining it professionally in a DBMS is being widely accepted. Many end users simply use applications written by database application programmers (see below) and so require little technical knowledge about DBMS software. Of course, sophisticated users who make more extensive use of a DBMS, such as writing their own queries, require a deeper understanding of its features.

In addition to end users and implementors, two other classes of people are associated with a DBMS: *application programmers* and *database administrators.*

Database application programmers develop packages that facilitate data access for end users, who are usually not computer professionals, using the host or data languages and software tools that DBMS vendors provide. (Such tools include report writers, spreadsheets, statistical packages, and the like.) Application programs should ideally access data through the external schema. It is possible to write applications that access data at a lower level, but such applications would compromise data independence.

A personal database is typically maintained by the individual who owns it and uses it. However, corporate or enterprise-wide databases are typically important enough and complex enough that the task of designing and maintaining the database is entrusted to a professional, called the **database administrator (DBA)**. The DBA is responsible for many critical tasks:

- **Design of the Conceptual and Physical Schemas:** The DBA is responsible for interacting with the users of the system to understand what data is to be stored in the DBMS and how it is likely to be used. Based on this knowledge, the DBA must design the conceptual schema (decide what relations to store) and the physical schema (decide how to store them). The DBA may also design widely used portions of the external schema, although users probably augment this schema by creating additional views.
- **Security and Authorization:** The DBA is responsible for ensuring that unauthorized data access is not permitted. In general, not everyone should be able to access all the data. In a relational DBMS, users can be granted permission to access only certain views and relations. For example, although you might allow students to find out course enrollments and who teaches a given course, you would not want students to see faculty salaries or each other's grade information. The DBA can enforce this policy by giving students permission to read only the Courseinfo view.
- **Data Availability and Recovery from Failures:** The DBA must take steps to ensure that if the system fails, users can continue to access as much of the uncorrupted data as possible. The DBA must also work to restore the data to a consistent state. The DBMS provides software support for these functions, but the DBA is responsible for implementing procedures to back up the data periodically and maintain logs of system activity (to facilitate recovery from a crash).
- **Database Tuning:** Users' needs are likely to evolve with time. The DBA is responsible for modifying the database, in particular the conceptual and physical schemas, to ensure adequate performance as requirements change.

1.10 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the main benefits of using a DBMS to manage data in applications involving extensive data access? **(Sections 1.1, 1.4)**
- When would you store data in a DBMS instead of in operating system files and vice-versa? **(Section 1.3)**

- What is a data model? What is the relational data model? What is data independence and how does a DBMS support it? **(Section 1.5)**
- Explain the advantages of using a query language instead of custom programs to process data. **(Section 1.6)**
- What is a transaction? What guarantees does a DBMS offer with respect to transactions? **(Section 1.7)**
- What are locks in a DBMS, and why are they used? What is write-ahead logging, and why is it used? What is checkpointing and why is it used? **(Section 1.7)**
- Identify the main components in a DBMS and briefly explain what they do. **(Section 1.8)**
- Explain the different roles of database administrators, application programmers, and end users of a database. Who needs to know the most about database systems? **(Section 1.9)**

EXERCISES

Exercise 1.1 Why would you choose a database system instead of simply storing data in operating system files? When would it make sense *not* to use a database system?

Exercise 1.2 What is logical data independence and why is it important?

Exercise 1.3 Explain the difference between logical and physical data independence.

Exercise 1.4 Explain the difference between external, internal, and conceptual schemas. How are these different schema layers related to the concepts of logical and physical data independence?

Exercise 1.5 What are the responsibilities of a DBA? If we assume that the DBA is never interested in running his or her own queries, does the DBA still need to understand query optimization? Why?

Exercise 1.6 Scrooge McNugget wants to store information (names, addresses, descriptions of embarrassing moments, etc.) about the many ducks on his payroll. Not surprisingly, the volume of data compels him to buy a database system. To save money, he wants to buy one with the fewest possible features, and he plans to run it as a stand-alone application on his PC clone. Of course, Scrooge does not plan to share his list with anyone. Indicate which of the following DBMS features Scrooge should pay for; in each case, also indicate why Scrooge should (or should not) pay for that feature in the system he buys.

1. A security facility.
2. Concurrency control.
3. Crash recovery.
4. A view mechanism.

5. A query language.

Exercise 1.7 Which of the following plays an important role in *representing* information about the real world in a database? Explain briefly.

1. The data definition language.
2. The data manipulation language.
3. The buffer manager.
4. The data model.

Exercise 1.8 Describe the structure of a DBMS. If your operating system is upgraded to support some new functions on OS files (e.g., the ability to force some sequence of bytes to disk), which layer(s) of the DBMS would you have to rewrite to take advantage of these new functions?

Exercise 1.9 Answer the following questions:

1. What is a transaction?
2. Why does a DBMS interleave the actions of different transactions instead of executing transactions one after the other?
3. What must a user guarantee with respect to a transaction and database consistency? What should a DBMS guarantee with respect to concurrent execution of several transactions and database consistency?
4. Explain the strict two-phase locking protocol.
5. What is the WAL property, and why is it important?

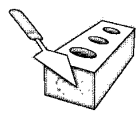
PROJECT-BASED EXERCISES

Exercise 1.10 Use a Web browser to look at the HTML documentation for Minibase. Try to get a feel for the overall architecture.

BIBLIOGRAPHIC NOTES

The evolution of database management systems is traced in [289]. The use of data models for describing real-world data is discussed in [423], and [425] contains a taxonomy of data models. The three levels of abstraction were introduced in [186, 712]. The network data model is described in [186], and [775] discusses several commercial systems based on this model. [721] contains a good annotated collection of systems-oriented papers on database management.

Other texts covering database management systems include [204, 245, 305, 339, 475, 574, 689, 747, 762]. [204] provides a detailed discussion of the relational model from a conceptual standpoint and is notable for its extensive annotated bibliography. [574] presents a performance-oriented perspective, with references to several commercial systems. [245] and [689] offer broad coverage of database system concepts, including a discussion of the hierarchical and network data models. [339] emphasizes the connection between database query languages and logic programming. [762] emphasizes data models. Of these texts, [747] provides the most detailed discussion of theoretical issues. Texts devoted to theoretical aspects include [3, 45, 501]. Handbook [744] includes a section on databases that contains introductory survey articles on a number of topics.



2

INTRODUCTION TO DATABASE DESIGN

- What are the steps in designing a database?
- Why is the ER model used to create an initial design?
- What are the main concepts in the ER model?
- What are guidelines for using the ER model effectively?
- How does database design fit within the overall design framework for complex software within large enterprises?
- What is UML and how is it related to the ER model?
- **Key concepts:** database design, conceptual, logical, and physical design; entity-relationship (ER) model, entity set, relationship set, attribute, instance, key; integrity constraints, one-to-many and many-to-many relationships, participation constraints; weak entities, class hierarchies, aggregation; UML, class diagrams, database diagrams, component diagrams.

> The great successful men of the world have used their imaginations. They think ahead and create their mental picture, and then go to work materializing that picture in all its details, filling in here, adding a little there, altering this bit and that bit, but steadily building, steadily building.
>
> —Robert Collier

The *entity-relationship (ER) data model* allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. It provides useful concepts that allow us to move from an informal description of what users want from

their database to a more detailed, precise description that can be implemented in a DBMS. In this chapter, we introduce the ER model and discuss how its features allow us to model a wide range of data faithfully.

We begin with an overview of database design in Section 2.1 in order to motivate our discussion of the ER model. Within the larger context of the overall design process, the ER model is used in a phase called *conceptual database design.* We then introduce the ER model in Sections 2.2, 2.3, and 2.4. In Section 2.5, we discuss database design issues involving the ER model. We briefly discuss conceptual database design for large enterprises in Section 2.6. In Section 2.7, we present an overview of UML, a design and modeling approach that is more general in its scope than the ER model.

In Section 2.8, we introduce a case study that is used as a running example throughout the book. The case study is an end-to-end database design for an Internet shop. We illustrate the first two steps in database design (requirements analysis and conceptual design) in Section 2.8. In later chapters, we extend this case study to cover the remaining steps in the design process.

We note that many variations of ER diagrams are in use and no widely accepted standards prevail. The presentation in this chapter is representative of the family of ER models and includes a selection of the most popular features.

2.1 DATABASE DESIGN AND ER DIAGRAMS

We begin our discussion of database design by observing that this is typically just one part, although a central part in data-intensive applications, of a larger software system design. Our primary focus is the design of the database, however, and we will not discuss other aspects of software design in any detail. We revisit this point in Section 2.7.

The database design process can be divided into six steps. The ER model is most relevant to the first three steps.

1. **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database. This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on.

Database Design Tools: Design tools are available from RDBMS vendors as well as third-party vendors. For example, see the following link for details on design and analysis tools from Sybase:
`http://www.sybase.com/products/application_tools`
The following provides details on Oracle's tools:
`http://www.oracle.com/tools`

Several methodologies have been proposed for organizing and presenting the information gathered in this step, and some automated tools have been developed to support this process.

2. **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints known to hold over this data. This step is often carried out using the ER model and is discussed in the rest of this chapter. The ER model is one of several high-level, or **semantic**, data models used in database design. The goal is to create a simple description of the data that closely matches how users and developers think of the data (and the people and processes to be represented in the data). This facilitates discussion among all the people involved in the design process, even those who have no technical background. At the same time, the initial design must be sufficiently precise to enable a straightforward translation into a data model supported by a commercial database system (which, in practice, means the relational model).

3. **Logical Database Design:** We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will consider only relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema. We discuss this step in detail in Chapter 3; the result is a conceptual schema, sometimes called the **logical schema**, in the relational data model.

2.1.1 Beyond ER Design

The ER diagram is just an approximate description of the data, constructed through a subjective evaluation of the information collected during requirements analysis. A more careful analysis can often refine the logical schema obtained at the end of Step 3. Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described next:

4. **Schema Refinement:** The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory. We discuss the theory of *normalizing* relations—restructuring them to ensure some desirable properties—in Chapter 19.

5. **Physical Database Design:** In this step, we consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps. We discuss physical design and database tuning in Chapter 20.

6. **Application and Security Design:** Any software project that involves a DBMS must consider aspects of the application that go beyond the database itself. Design methodologies like UML (Section 2.7) try to address the complete software design and development cycle. Briefly, we must identify the entities (e.g., users, user groups, departments) and processes involved in the application. We must describe the role of each entity in every process that is reflected in some application task, as part of a complete workflow for that task. For each role, we must identify the parts of the database that must be accessible and the parts of the database that must *not* be accessible, and we must take steps to ensure that these access rules are enforced. A DBMS provides several mechanisms to assist in this step, and we discuss this in Chapter 21.

In the implementation phase, we must code each task in an application language (e.g., Java), using the DBMS to access data. We discuss application development in Chapters 6 and 7.

In general, our division of the design process into steps should be seen as a classification of the *kinds* of steps involved in design. Realistically, although we might begin with the six step process outlined here, a complete database design will probably require a subsequent **tuning** phase in which all six kinds of design steps are interleaved and repeated until the design is satisfactory.

2.2 ENTITIES, ATTRIBUTES, AND ENTITY SETS

An **entity** is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the man-

ager of the toy department. It is often useful to identify a collection of similar entities. Such a collection is called an **entity set**. Note that entity sets need not be disjoint; the collection of toy department employees and the collection of appliance department employees may both contain employee John Doe (who happens to work in both departments). We could also define an entity set called Employees that contains both the toy and appliance department employee sets.

An entity is described using a set of **attributes**. All entities in a given entity set have the same attributes; this is what we mean by *similar*. (This statement is an oversimplification, as we will see when we discuss inheritance hierarchies in Section 2.4.4, but it suffices for now and highlights the main idea.) Our choice of attributes reflects the level of detail at which we wish to represent information about entities. For example, the Employees entity set could use name, social security number (ssn), and parking lot (lot) as attributes. In this case we will store the name, social security number, and lot number for each employee. However, we will not store, say, an employee's address (or gender or age).

For each attribute associated with an entity set, we must identify a **domain** of possible values. For example, the domain associated with the attribute *name* of Employees might be the set of 20-character strings.[1] As another example, if the company rates employees on a scale of 1 to 10 and stores ratings in a field called *rating*, the associated domain consists of integers 1 through 10. Further, for each entity set, we choose a *key*. A **key** is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one **candidate** key; if so, we designate one of them as the **primary** key. For now we assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set; that is, the set of attributes contains a key. We revisit this point in Section 2.4.3.

The Employees entity set with attributes *ssn, name,* and *lot* is shown in Figure 2.1. An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the primary key is underlined. The domain information could be listed along with the attribute name, but we omit this to keep the figures compact. The key is *ssn*.

2.3 RELATIONSHIPS AND RELATIONSHIP SETS

A **relationship** is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department.

[1]To avoid confusion, we assume that attribute names do not repeat across entity sets. This is not a real limitation because we can always use the entity set name to resolve ambiguities if the same attribute name is used in more than one entity set.

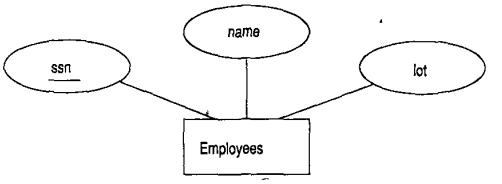


Figure 2.1 The Employees Entity Set

As with entities, we may wish to collect a set of similar relationships into a **relationship set**. A relationship set can be thought of as a set of n-tuples:

$$\{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$

Each n-tuple denotes a relationship involving n entities e_1 through e_n, where entity e_i is in entity set E_i. In Figure 2.2 we show the relationship set Works_In, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.

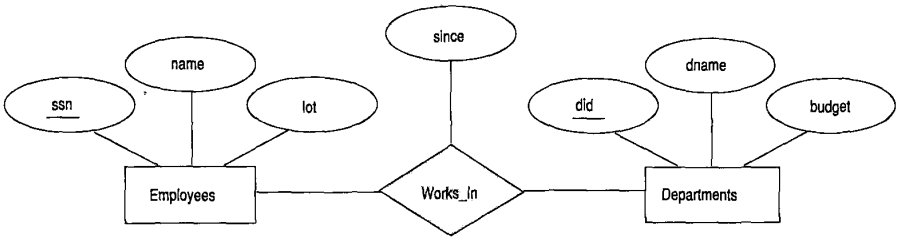


Figure 2.2 The Works_In Relationship Set

A relationship can also have **descriptive attributes**. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured in Figure 2.2 by adding an attribute, *since*, to Works_In. A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. In the Works_In relationship set, for example, each Works_In relationship must be uniquely identified by the combination of employee *ssn* and department *did*. Thus, for a given employee-department pair, we cannot have more than one associated *since* value.

An **instance** of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a 'snapshot' of the relationship set at some instant

in time. An instance of the Works_In relationship set is shown in Figure 2.3. Each Employees entity is denoted by its *ssn*, and each Departments entity is denoted by its *did*, for simplicity. The *since* value is shown beside each relationship. (The 'many-to-many' and 'total participation' comments in the figure are discussed later, when we discuss integrity constraints.)

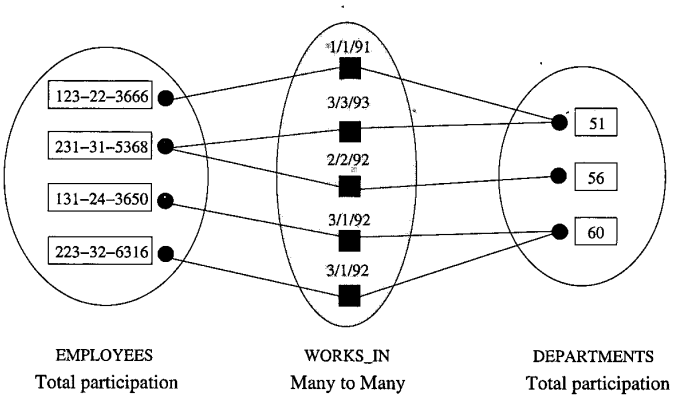


Figure 2.3 An Instance of the Works_In Relationship Set

As another example of an ER diagram, suppose that each department has offices in several locations and we want to record the locations at which each employee works. This relationship is **ternary** because we must record an association between an employee, a department, and a location. The ER diagram for this variant of Works_In, which we call Works_In2, is shown in Figure 2.4.

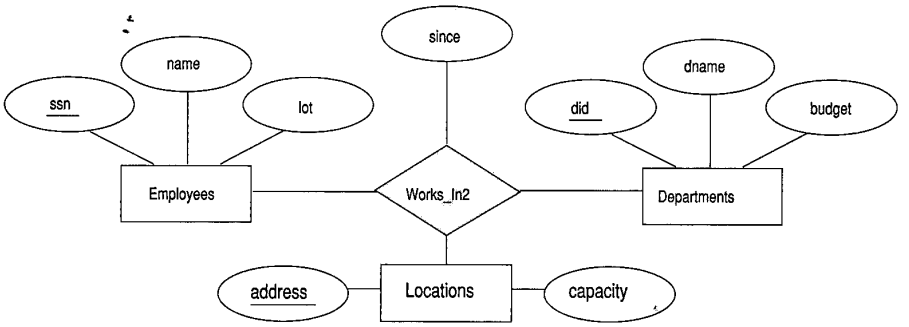


Figure 2.4 A Ternary Relationship Set

The entity sets that participate in a relationship set need not be distinct; sometimes a relationship might involve two entities in the same entity set. For example, consider the Reports_To relationship set shown in Figure 2.5. Since

employees report to other employees, every relationship in Reports_To is of the form (emp_1, emp_2), where both emp_1 and emp_2 are entities in Employees. However, they play different **roles**: emp_1 reports to the managing employee emp_2, which is reflected in the **role indicators** *supervisor* and *subordinate* in Figure 2.5. If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports_To relationship set has attributes corresponding to the *ssn* of the supervisor and the *ssn* of the subordinate, and the names of these attributes are *supervisor_ssn* and *subordinate_ssn*.

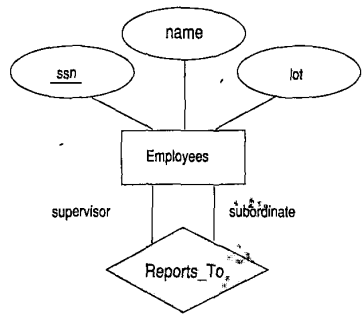


Figure 2.5 The Reports_To Relationship Set

2.4 ADDITIONAL FEATURES OF THE ER MODEL

We now look at some of the constructs in the ER model that allow us to describe some subtle properties of the data. The expressiveness of the ER model is a big reason for its widespread use.

2.4.1 Key Constraints

Consider the Works_In relationship shown in Figure 2.2. An employee can work in several departments, and a department can have several employees, as illustrated in the Works_In instance shown in Figure 2.3. Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees.

Now consider another relationship set called Manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department has at most one manager is

an example of a **key constraint**, and it implies that each Departments entity appears in at most one Manages relationship in any allowable instance of Manages. This restriction is indicated in the ER diagram of Figure 2.6 by using an arrow from Departments to Manages. Intuitively, the arrow states that given a Departments entity, we can uniquely determine the Manages relationship in which it appears.

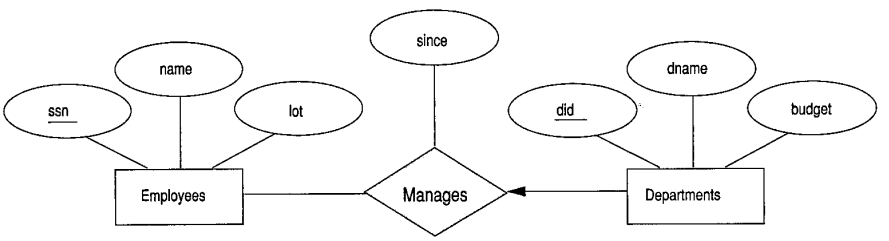


Figure 2.6 Key Constraint on Manages

An instance of the Manages relationship set is shown in Figure 2.7. While this is also a potential instance for the Works_In relationship set, the instance of Works_In shown in Figure 2.3 violates the key constraint on Manages.

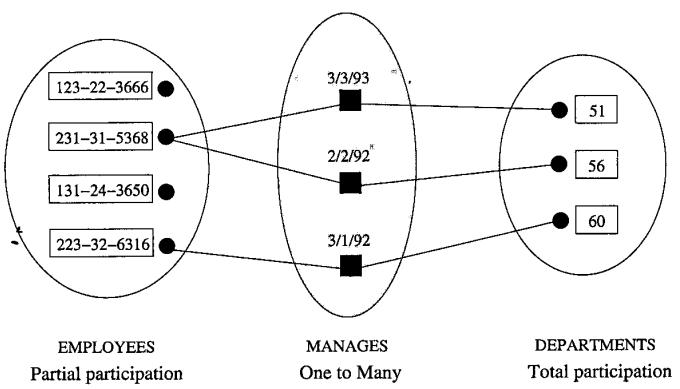


Figure 2.7 An Instance of the Manages Relationship Set

A relationship set like Manages is sometimes said to be **one-to-many**, to indicate that *one* employee can be associated with *many* departments (in the capacity of a manager), whereas each department can be associated with at most one employee as its manager. In contrast, the Works_In relationship set, in which an employee is allowed to work in several departments and a department is allowed to have several employees, is said to be **many-to-many**.

If we add the restriction that each employee can manage at most one department to the Manages relationship set, which would be indicated by adding an arrow from Employees to Manages in Figure 2.6, we have a **one-to-one** relationship set.

Key Constraints for Ternary Relationships

We can extend this convention—and the underlying key constraint concept—to relationship sets involving three or more entity sets: If an entity set E has a key constraint in a relationship set R, each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R. To indicate a key constraint on entity set E in relationship set R, we draw an arrow from E to R.

In Figure 2.8, we show a ternary relationship with key constraints. Each employee works in at most one department and at a single location. An instance of the Works_In3 relationship set is shown in Figure 2.9. Note that each department can be associated with several employees and locations and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

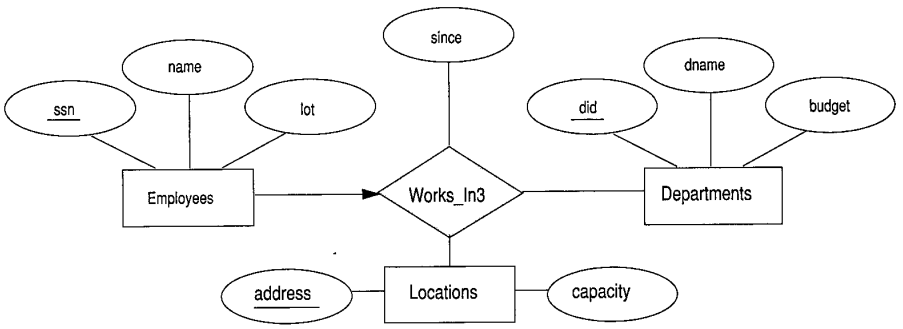


Figure 2.8 A Ternary Relationship Set with Key Constraints

2.4.2 Participation Constraints

The key constraint on Manages tells us that a department has at most one manager. A natural question to ask is whether every department has a manager. Let us say that every department is required to have a manager. This requirement is an example of a **participation constraint**; the participation of the entity set Departments in the relationship set Manages is said to be **total**. A participation that is not total is said to be **partial**. As an example, the

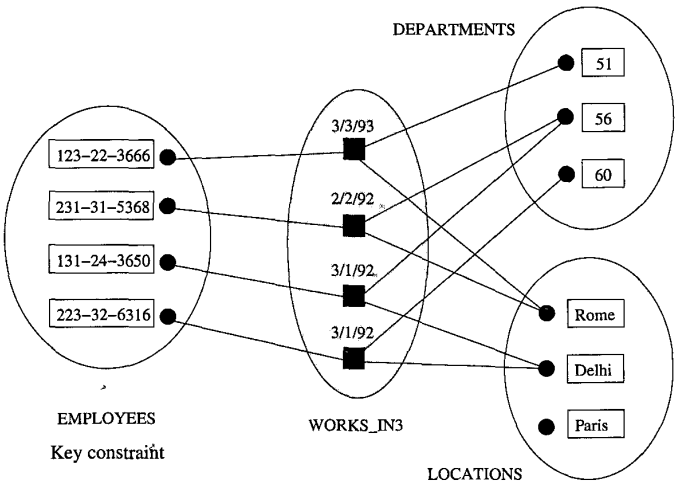


Figure 2.9 An Instance of Works_In3

participation of the entity set Employees in Manages is partial, since not every employee gets to manage a department.

Revisiting the Works_In relationship set, it is natural to expect that each employee works in at least one department and that each department has at least one employee. This means that the participation of both Employees and Departments in Works_In is total. The ER diagram in Figure 2.10 shows both the Manages and Works_In relationship sets and all the given constraints. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint. The instances of Works_In and Manages shown in Figures 2.3 and 2.7 satisfy all the constraints in Figure 2.10.

2.4.3 Weak Entities

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

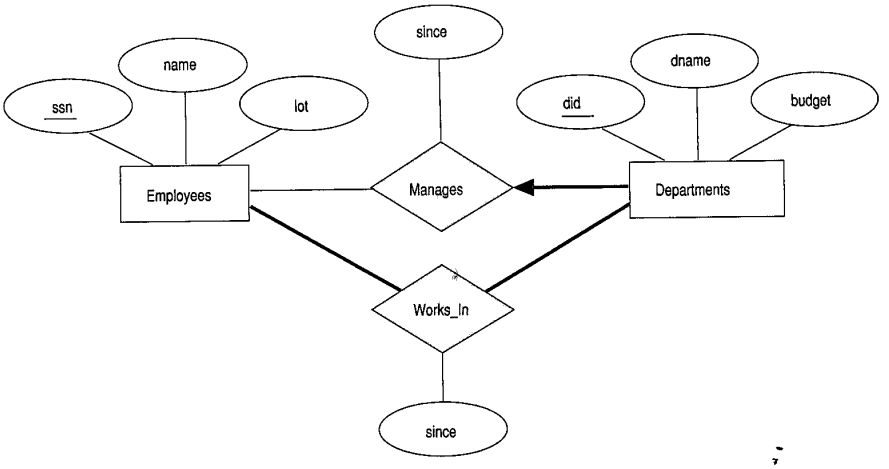


Figure 2.10 Manages and Works_In

We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be *pname* and *age*. The attribute *pname* does *not* identify a dependent uniquely. Recall that the key for Employees is *ssn*; thus we might have two employees called Smethurst and each might have a son called Joe.

Dependents is an example of a **weak entity set**. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**.

The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set.

For example, a Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity. The set of attributes of a weak entity set that uniquely identify a weak entity for a given owner entity is called a *partial key* of the weak entity set. In our example, *pname* is a partial key for Dependents.

The Dependents weak entity set and its relationship to Employees is shown in Figure 2.11. The total participation of Dependents in Policy is indicated by linking them with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that *pname* is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same *pname* value.

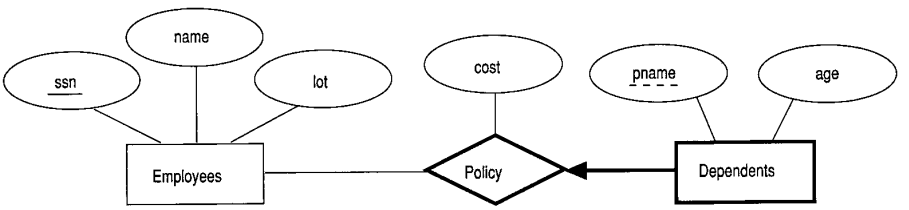


Figure 2.11 A Weak Entity Set

2.4.4 Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an Hourly_Emps entity set and a Contract_Emps entity set to distinguish the basis on which they are paid. We might have attributes *hours_worked* and *hourly_wage* defined for Hourly_Emps and an attribute *contractid* defined for Contract_Emps.

We want the semantics that every entity in one of these sets is also an Employees entity and, as such, must have all the attributes of Employees defined. Therefore, the attributes defined for an Hourly_Emps entity are the attributes for Employees plus Hourly_Emps. We say that the attributes for the entity set Employees are **inherited** by the entity set Hourly_Emps and that Hourly_Emps **ISA** (read *is a*) Employees. In addition—and in contrast to class hierarchies in programming languages such as C++—there is a constraint on queries over instances of these entity sets: A query that asks for all Employees entities must consider all Hourly_Emps and Contract_Emps entities as well. Figure 2.12 illustrates the class hierarchy.

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees as Senior_Emps. We can modify Figure 2.12 to reflect this change by adding a second ISA node as a child of Employees and making Senior_Emps a child of this node. Each of these entity sets might be classified further, creating a multilevel ISA hierarchy.

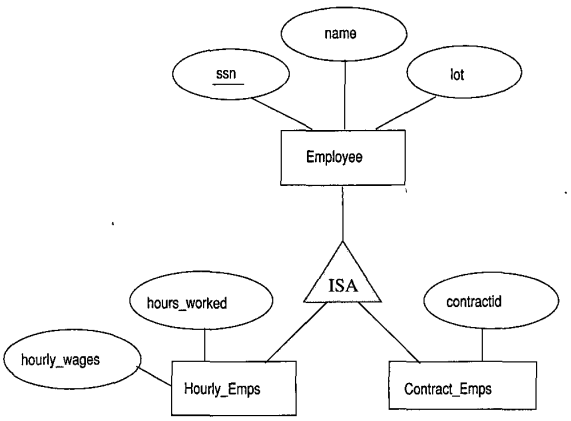


Figure 2.12 Class Hierarchy

A class hierarchy can be viewed in one of two ways:

- Employees is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically, the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.
- Hourly_Emps and Contract_Emps are **generalized** by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor_Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically, the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

We can specify two kinds of constraints with respect to ISA hierarchies, namely, *overlap* and *covering* constraints. **Overlap constraints** determine whether two subclasses are allowed to contain the same entity. For example, can Attishoo be both an Hourly_Emps entity and a Contract_Emps entity? Intuitively, no. Can he be both a Contract_Emps entity and a Senior_Emps entity? Intuitively, yes. We denote this by writing 'Contract_Emps `OVERLAPS` Senior_Emps.' In the absence of such a statement, we assume by default that entity sets are constrained to have no overlap.

Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass. For example, does every Employees

entity have to belong to one of its subclasses? Intuitively, no. Does every Motor_Vehicles entity have to be either a Motorboats entity or a Cars entity? Intuitively, yes; a characteristic property of generalization hierarchies is that every instance of a superclass is an instance of a subclass. We denote this by writing 'Motorboats AND Cars COVER Motor_Vehicles.' In the absence of such a statement, we assume by default that there is no covering constraint; we can have motor vehicles that are not motorboats or cars.

There are two basic reasons for identifying subclasses (by specialization or generalization):

1. We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, *hourly_wages* does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.
2. We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior_Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

2.4.5 Aggregation

As defined thus far, a relationship set is an association between entity sets. Sometimes, we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

To define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation*. **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

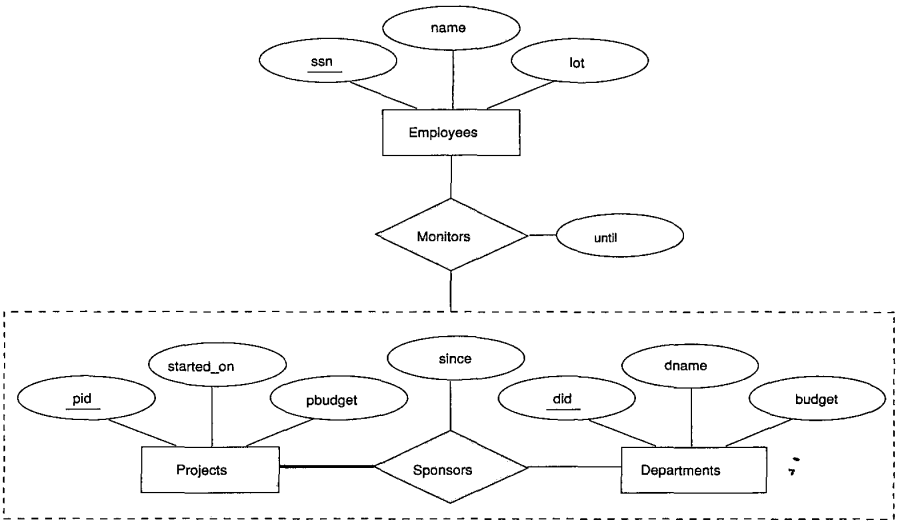


Figure 2.13 Aggregation

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can we not express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the Monitors relationship has an attribute *until* that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute *since* of Sponsors, which is the date when the sponsorship took effect. The use of aggregation versus a ternary relationship may also be guided by certain integrity constraints, as explained in Section 2.5.4.

2.5 CONCEPTUAL DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.

2.5.1 Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as "Find all employees with an address in Madison, WI."

For another example of when to model a concept as an entity set rather than an attribute, consider the relationship set (called Works_In4) shown in Figure 2.14.

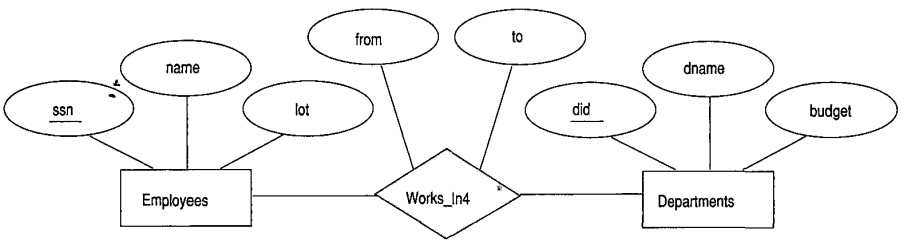


Figure 2.14 The Works_In4 Relationship Set

It differs from the Works_In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics, because a relationship is uniquely identified by the participating entities (recall from Section

2.3). The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.

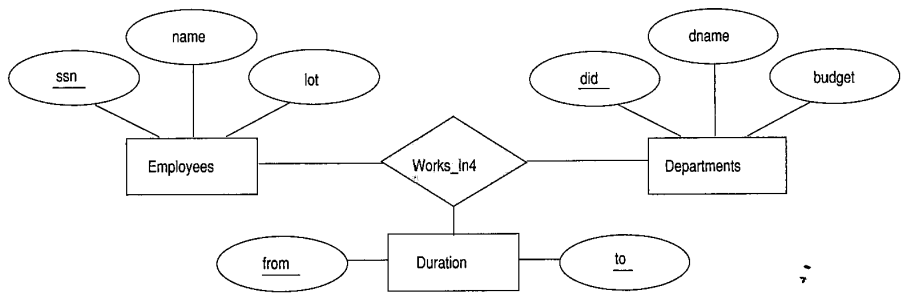


Figure 2.15 The Works_In4 Relationship Set

In some versions of the ER model, attributes are allowed to take on sets as values. Given this feature, we could make Duration an attribute of Works_In, rather than an entity set; associated with each Works_In relationship, we would have a set of intervals. This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

2.5.2 Entity versus Relationship

Consider the relationship set called Manages in Figure 2.6. Suppose that each department manager is given a discretionary budget (*dbudget*), as shown in Figure 2.16, in which we have also renamed the relationship set to Manages2.

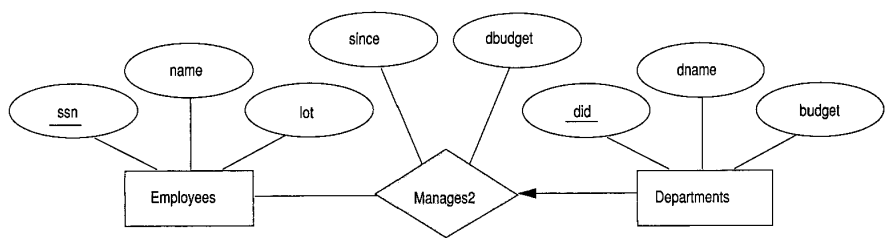


Figure 2.16 Entity versus Relationship

Given a department, we know the manager, as well as the manager's starting date and budget for that department. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case, each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field, leading to redundant storage of the same information. Another problem with this design is that it is misleading; it suggests that the budget is associated with the relationship, when it is actually associated with the manager.

We can address these problems by introducing a new entity set called Managers (which can be placed below Employees in an ISA hierarchy, to show that every manager is also an employee). The attributes *since* and *dbudget* now describe a manager entity, as intended. As a variation, while every manager has a budget, each manager may have a different starting date (as manager) for each department. In this case *dbudget* is an attribute of Managers, but *since* is an attribute of the relationship set between managers and departments.

The imprecise nature of ER modeling can thus make it difficult to recognize underlying entities, and we might associate attributes with relationships rather than the appropriate entities. In general, such mistakes lead to redundant storage of the same information and can cause many problems. We discuss redundancy and its attendant problems in Chapter 19, and present a technique called *normalization* to eliminate redundancies from tables.

2.5.3 Binary versus Ternary Relationships

Consider the ER diagram shown in Figure 2.17. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking *pname* in conjunction with the *policyid* of a policy entity (which, intuitively, covers the given dependent).

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a

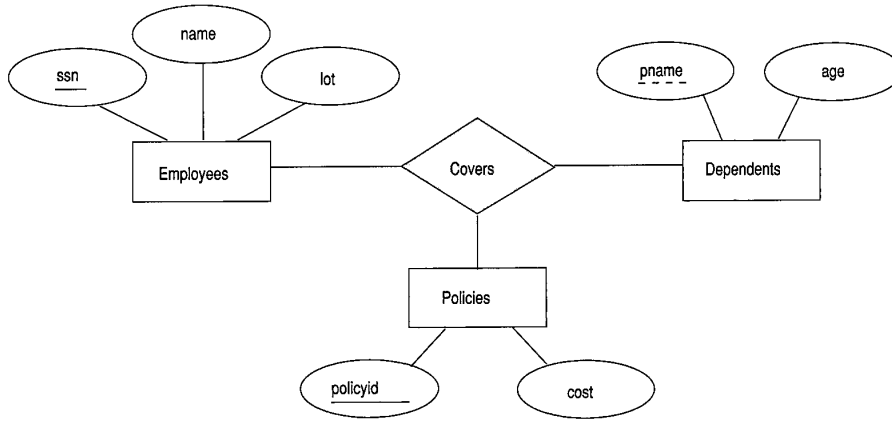


Figure 2.17 Policies as an Entity Set

policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third requirement, the best way to model this situation is to use two binary relationships, as shown in Figure 2.18.

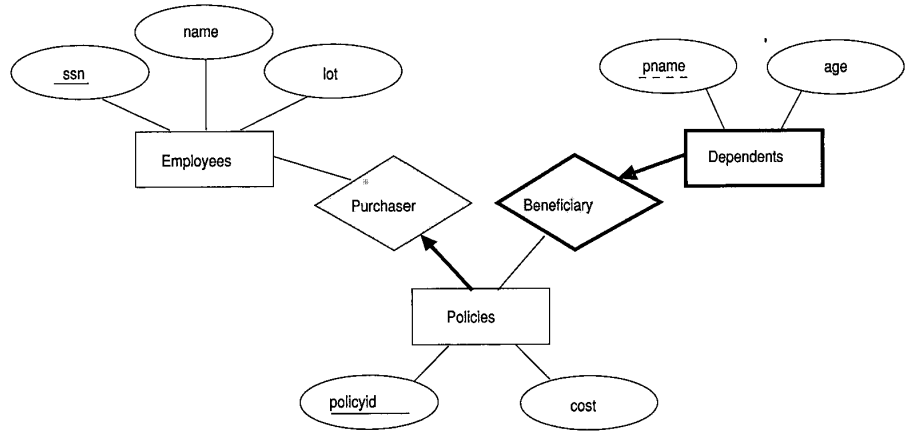


Figure 2.18 Policy Revisited

This example really has two relationships involving Policies, and our attempt to use a single ternary relationship (Figure 2.17) is inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figures 2.4 and 2.15.

As a typical example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute *qty*) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier 'can supply' certain parts, that a department 'needs' some parts, or that a department 'deals with' a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

- The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
- We cannot represent the *qty* attribute of a contract cleanly.

2.5.4 Aggregation versus Ternary Relationships

As we noted in Section 2.4.5, the choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a *relationship set* to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in Figure 2.13. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don't need to record the *until* attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure 2.19.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship Monitors in Figure 2.13. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.

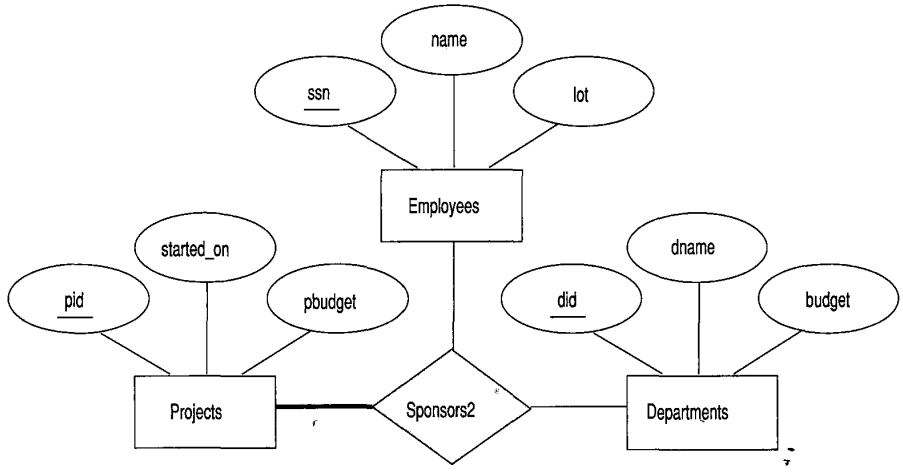


Figure 2.19 Using a Ternary Relationship instead of Aggregation

2.6 CONCEPTUAL DESIGN FOR LARGE ENTERPRISES

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups. Using a high-level, semantic data model, such as ER diagrams, for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and easily understood by the many people who must provide input to the design process.

An important aspect of the design process is the methodology used to structure the development of the overall design and ensure that the design takes into account all user requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

An alternative approach is to develop separate conceptual schemas for different user groups and then *integrate* these conceptual schemas. To integrate multi-

ple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, differences in measurement units). This task is difficult in its own right. In some situations, schema integration cannot be avoided; for example, when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to *heterogeneous* data sources, often maintained by different organizations.

2.7 THE UNIFIED MODELING LANGUAGE

There are many approaches to end-to-end software system design, covering all the steps from identifying the business requirements to the final specifications for a complete application, including workflow, user interfaces, and many aspects of software systems that go well beyond databases and the data stored in them. In this section, we briefly discuss an approach that is becoming popular, called the **unified modeling language (UML) approach**.

UML, like the ER model, has the attractive feature that its constructs can be drawn as diagrams. It encompasses a broader spectrum of the software design process than the ER model:

- **Business Modeling:** In this phase, the goal is to describe the business processes involved in the software application being developed.
- **System Modeling:** The understanding of business processes is used to identify the requirements for the software application. One part of the requirements is the database requirements.
- **Conceptual Database Modeling:** This step corresponds to the creation of the ER design for the database. For this purpose, UML provides many constructs that parallel the ER constructs.
- **Physical Database Modeling:** UML also provides pictorial representations for physical database design choices, such as the creation of table spaces and indexes. (We discuss physical database design in later chapters, but not the corresponding UML constructs.)
- **Hardware System Modeling:** UML diagrams can be used to describe the hardware configuration used for the application.

There are many kinds of diagrams in UML. **Use case** diagrams describe the actions performed by the system in response to user requests, and the people involved in these actions. These diagrams specify the external functionality that the system is expected to support.

Activity diagrams show the flow of actions in a business process. **Statechart** diagrams describe dynamic interactions between system objects. These diagrams, used in business and system modeling, describe how the external functionality is to be implemented, consistent with the business rules and processes of the enterprise.

Class diagrams are similar to ER diagrams, although they are more general in that they are intended to model *application* entities (intuitively, important program components) and their logical relationships in addition to data entities and their relationships.

Both entity sets and relationship sets can be represented as classes in UML, together with key constraints, weak entities, and class hierarchies. The term *relationship* is used slightly differently in UML, and UML's relationships are binary. This sometimes leads to confusion over whether relationship sets in an ER diagram involving three or more entity sets can be directly represented in UML. The confusion disappears once we understand that all relationship sets (in the ER sense) are represented as classes in UML; the binary UML 'relationships' are essentially just the links shown in ER diagrams between entity sets and relationship sets.

Relationship sets with key constraints are usually omitted from UML diagrams, and the relationship is indicated by directly linking the entity sets involved. For example, consider Figure 2.6. A UML representation of this ER diagram would have a class for Employees, a class for Departments, and the relationship Manages is shown by linking these two classes. The link can be labeled with a name and cardinality information to show that a department can have only one manager.

As we will see in Chapter 3, ER diagrams are translated into the relational model by mapping each entity set into a table and each relationship set into a table. Further, as we will see in Section 3.5.3, the table corresponding to a one-to-many relationship set is typically omitted by including some additional information about the relationship in the table for one of the entity sets involved. Thus, UML class diagrams correspond closely to the tables created by mapping an ER diagram.

Indeed, every class in a UML class diagram is mapped into a table in the corresponding UML database diagram. UML's **database diagrams** show how classes are represented in the database and contain additional details about the structure of the database such as integrity constraints and indexes. Links (UML's 'relationships') between UML classes lead to various integrity constraints between the corresponding tables. Many details specific to the relational model (e.g., *views*, *foreign keys*, *null-allowed fields*) and that reflect

physical design choices (e.g., indexed fields) can be modeled in UML database diagrams.

UML's **component** diagrams describe storage aspects of the database, such as *tablespaces* and *database partitions*), as well as interfaces to applications that access the database. Finally, **deployment** diagrams show the hardware aspects of the system.

Our objective in this book is to concentrate on the data stored in a database and the related design issues. To this end, we deliberately take a simplified view of the other steps involved in software design and development. Beyond the specific discussion of UML, the material in this section is intended to place the design issues that we cover within the context of the larger software design process. We hope that this will assist readers interested in a more comprehensive discussion of software design to complement our discussion by referring to other material on their preferred approach to overall system design.

2.8 CASE STUDY: THE INTERNET SHOP

We now introduce an illustrative, 'cradle-to-grave' design case study that we use as a running example throughout this book. DBDudes Inc., a well-known database consulting firm, has been called in to help Barns and Nobble (B&N) with its database design and implementation. B&N is a large bookstore specializing in books on horse racing, and it has decided to go online. DBDudes first verifies that B&N is willing and able to pay its steep fees and then schedules a lunch meeting—billed to B&N, naturally—to do requirements analysis.

2.8.1 Requirements Analysis

The owner of B&N, unlike many people who need a database, has thought extensively about what he wants and offers a concise summary:

"I would like my customers to be able to browse my catalog of books and place orders over the Internet. Currently, I take orders over the phone. I have mostly corporate customers who call me and give me the ISBN number of a book and a quantity; they often pay by credit card. I then prepare a shipment that contains the books they ordered. If I don't have enough copies in stock, I order additional copies and delay the shipment until the new copies arrive; I want to ship a customer's entire order together. My catalog includes all the books I sell. For each book, the catalog contains its ISBN number, title, author, purchase price, sales price, and the year the book was published. Most of my customers are regulars, and I have records with their names and addresses.

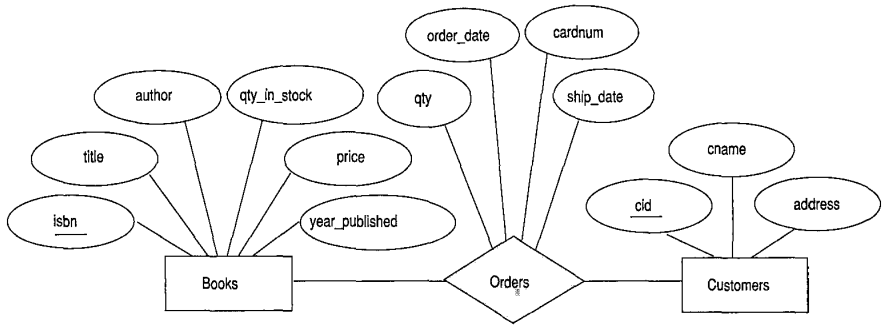


Figure 2.20 ER Diagram of the Initial Design

New customers have to call me first and establish an account before they can use my website.

On my new website, customers should first identify themselves by their unique customer identification number. Then they should be able to browse my catalog and to place orders online."

DBDudes's consultants are a little surprised by how quickly the requirements phase is completed—it usually takes weeks of discussions (and many lunches and dinners) to get this done—but return to their offices to analyze this information.

2.8.2 Conceptual Design

In the conceptual design step, DBDudes develops a high level description of the data in terms of the ER model. The initial design is shown in Figure 2.20. Books and customers are modeled as entities and related through orders that customers place. Orders is a relationship set connecting the Books and Customers entity sets. For each order, the following attributes are stored: quantity, order date, and ship date. As soon as an order is shipped, the ship date is set; until then the ship date is set to *null*, indicating that this order has not been shipped yet.

DBDudes has an internal design review at this point, and several questions are raised. To protect their identities, we will refer to the design team leader as Dude 1 and the design reviewer as Dude 2.

Dude 2: What if a customer places two orders for the same book in one day?
Dude 1: The first order is handled by creating a new Orders relationship and

the second order is handled by updating the value of the quantity attribute in this relationship.
Dude 2: What if a customer places two orders for different books in one day?
Dude 1: No problem. Each instance of the Orders relationship set relates the customer to a different book.
Dude 2: Ah, but what if a customer places two orders for the same book on different days?
Dude 1: We can use the attribute order date of the orders relationship to distinguish the two orders.
Dude 2: Oh no you can't. The attributes of Customers and Books must jointly contain a key for Orders. So this design does not allow a customer to place orders for the same book on different days.
Dude 1: Yikes, you're right. Oh well, B&N probably won't care; we'll see.

DBDudes decides to proceed with the next phase, logical database design; we rejoin them in Section 3.8.

2.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Name the main steps in database design. What is the goal of each step? In which step is the ER model mainly used? **(Section 2.1)**
- Define these terms: *entity*, *entity set*, *attribute*, *key*. **(Section 2.2)**
- Define these terms: *relationship*, *relationship set*, *descriptive attributes*. **(Section 2.3)**
- Define the following kinds of constraints, and give an example of each: *key constraint*, *participation constraint*. What is a *weak entity*? What are *class hierarchies*? What is *aggregation*? Give an example scenario motivating the use of each of these ER model design constructs. **(Section 2.4)**
- What guidelines would you use for each of these choices when doing ER design: Whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. **(Section 2.5)**
- Why is designing a database for a large enterprise especially hard? **(Section 2.6)**
- What is UML? How does database design fit into the overall design of a data-intensive software system? How is UML related to ER diagrams? **(Section 2.7)**

EXERCISES

Exercise 2.1 Explain the following terms briefly: *attribute, domain, entity, relationship, entity set, relationship set, one-to-many relationship, many-to-many relationship, participation constraint, overlap constraint, covering constraint, weak entity set, aggregation*, and *role indicator.*

Exercise 2.2 A university database contains information about professors (identified by social security number, or SSN) and courses (identified by courseid). Professors teach courses; each of the following situations concerns the Teaches relationship set. For each situation, draw an ER diagram that describes it (assuming no further constraints hold).

1. Professors can teach the same course in several semesters, and each offering must be recorded.
2. Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded. (Assume this condition applies in all subsequent questions.)
3. Every professor must teach some course.
4. Every professor teaches exactly one course (no more, no less).
5. Every professor teaches exactly one course (no more, no less), and every course must be taught by some professor.
6. Now suppose that certain courses can be taught by a team of professors jointly, but it is possible that no one professor in a team can teach the course. Model this situation, introducing additional entity sets and relationship sets if necessary.

Exercise 2.3 Consider the following information about a university database:

- Professors have an SSN, a name, an age, a rank, and a research specialty.
- Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
- Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
- Each project is managed by one professor (known as the project's principal investigator).
- Each project is worked on by one or more professors (known as the project's co-investigators).
- Professors can manage and/or work on multiple projects.
- Each project is worked on by one or more graduate students (known as the project's research assistants).
- When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
- Departments have a department number, a department name, and a main office.
- Departments have a professor (known as the chairman) who runs the department.
- Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.
- Graduate students have one major department in which they are working on their degree.

- Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university. Use only the basic ER model here; that is, entities, relationships, and attributes. Be sure to indicate any key and participation constraints.

Exercise 2.4 A company database needs to store information about employees (identified by *ssn*, with *salary* and *phone* as attributes), departments (identified by *dno*, with *dname* and *budget* as attributes), and children of employees (with *name* and *age* as attributes). Employees *work* in departments; each department is *managed by* an employee; a child must be identified uniquely by *name* when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.

Exercise 2.5 Notown Records has decided to store information about musicians who perform on its albums (as well as other company data) in a database. The company has wisely chosen to hire you as a database designer (at your usual consulting fee of $2500/day).

- Each musician that records at Notown has an SSN, a name, an address, and a phone number. Poorly paid musicians often share the same address, and no address has more than one phone.
- Each instrument used in songs recorded at Notown has a name (e.g., guitar, synthesizer, flute) and a musical key (e.g., C, B-flat, E-flat).
- Each album recorded on the Notown label has a title, a copyright date, a format (e.g., CD or MC), and an album identifier.
- Each song recorded at Notown has a title and an author.
- Each musician may play several instruments, and a given instrument may be played by several musicians.
- Each album has a number of songs on it, but no song may appear on more than one album.
- Each song is performed by one or more musicians, and a musician may perform a number of songs.
- Each album has exactly one musician who acts as its producer. A musician may produce several albums, of course.

Design a conceptual schema for Notown and draw an ER diagram for your schema. The preceding information describes the situation that the Notown database must model. Be sure to indicate all key and cardinality constraints and any assumptions you make. Identify any constraints you are unable to capture in the ER diagram and briefly explain why you could not express them.

Exercise 2.6 Computer Sciences Department frequent fliers have been complaining to Dane County Airport officials about the poor organization at the airport. As a result, the officials decided that all information related to the airport should be organized using a DBMS, and you have been hired to design the database. Your first task is to organize the information about all the airplanes stationed and maintained at the airport. The relevant information is as follows:

- Every airplane has a registration number, and each airplane is of a specific model.
- The airport accommodates a number of airplane models, and each model is identified by a model number (e.g., DC-10) and has a capacity and a weight.
- A number of technicians work at the airport. You need to store the name, SSN, address, phone number, and salary of each technician.
- Each technician is an expert on one or more plane model(s), and his or her expertise may overlap with that of other technicians. This information about technicians must also be recorded.
- Traffic controllers must have an annual medical examination. For each traffic controller, you must store the date of the most recent exam.
- All airport employees (including technicians) belong to a union. You must store the union membership number of each employee. You can assume that each employee is uniquely identified by a social security number.
- The airport has a number of tests that are used periodically to ensure that airplanes are still airworthy. Each test has a Federal Aviation Administration (FAA) test number, a name, and a maximum possible score.
- The FAA requires the airport to keep track of each time a given airplane is tested by a given technician using a given test. For each testing event, the information needed is the date, the number of hours the technician spent doing the test, and the score the airplane received on the test.

1. Draw an ER diagram for the airport database. Be sure to indicate the various attributes of each entity and relationship set; also specify the key and participation constraints for each relationship set. Specify any necessary overlap and covering constraints as well (in English).
2. The FAA passes a regulation that tests on a plane must be conducted by a technician who is an expert on that model. How would you express this constraint in the ER diagram? If you cannot express it, explain briefly.

Exercise 2.7 The Prescriptions-R-X chain of pharmacies has offered to give you a free lifetime supply of medicine if you design its database. Given the rising cost of health care, you agree. Here's the information that you gather:

- Patients are identified by an SSN, and their names, addresses, and ages must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty, and years of experience must be recorded.
- Each pharmaceutical company is identified by name and has a phone number.
- For each drug, the trade name and formula must be recorded. Each drug is sold by a given pharmaceutical company, and the trade name identifies a drug uniquely from among the products of that company. If a pharmaceutical company is deleted, you need not keep track of its products any longer.
- Each pharmacy has a name, address, and phone number.
- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy sells several drugs and has a price for each. A drug could be sold at several pharmacies, and the price could vary from one pharmacy to another.

- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and a quantity associated with it. You can assume that, if a doctor prescribes the same drug for the same patient more than once, only the last such prescription needs to be stored.
- Pharmaceutical companies have long-term contracts with pharmacies. A pharmaceutical company can contract with several pharmacies, and a pharmacy can contract with several pharmaceutical companies. For each contract, you have to store a start date, an end date, and the text of the contract.
- Pharmacies appoint a supervisor for each contract. There must always be a supervisor for each contract, but the contract supervisor can change over the lifetime of the contract.

1. Draw an ER diagram that captures the preceding information. Identify any constraints not captured by the ER diagram.
2. How would your design change if each drug must be sold at a fixed price by all pharmacies?
3. How would your design change if the design requirements change as follows: If a doctor prescribes the same drug for the same patient more than once, several such prescriptions may have to be stored.

Exercise 2.8 Although you always wanted to be an artist, you ended up being an expert on databases because you love to cook data and you somehow confused *database* with *data baste*. Your old love is still there, however, so you set up a database company, ArtBase, that builds a product for art galleries. The core of this product is a database with a schema that captures all the information that galleries need to maintain. Galleries keep information about artists, their names (which are unique), birthplaces, age, and style of art. For each piece of artwork, the artist, the year it was made, its unique title, its type of art (e.g., painting, lithograph, sculpture, photograph), and its price must be stored. Pieces of artwork are also classified into groups of various kinds, for example, portraits, still lifes, works by Picasso, or works of the 19th century; a given piece may belong to more than one group. Each group is identified by a name (like those just given) that describes the group. Finally, galleries keep information about customers. For each customer, galleries keep that person's unique name, address, total amount of dollars spent in the gallery (very important!), and the artists and groups of art that the customer tends to like.

Draw the ER diagram for the database.

Exercise 2.9 Answer the following questions.

- Explain the following terms briefly: *UML, use case diagrams, statechart diagrams, class diagrams, database diagrams, component diagrams*, and *deployment diagrams*.
- Explain the relationship between ER diagrams and UML.

BIBLIOGRAPHIC NOTES

Several books provide a good treatment of conceptual design; these include [63] (which also contains a survey of commercial database design tools) and [730].

The ER model was proposed by Chen [172], and extensions have been proposed in a number of subsequent papers. Generalization and aggregation were introduced in [693]. [390, 589]

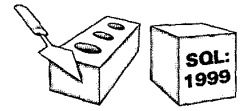
contain good surveys of semantic data models. Dynamic and temporal aspects of semantic data models are discussed in [749].

[731] discusses a design methodology based on developing an ER diagram and then translating it to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [513, 514].

The entity-relationship conference proceedings contain numerous papers on conceptual design, with an emphasis on the ER model; for example, [698].

The OMG home page (`www.omg.org`) contains the specification for UML and related modeling standards. Numerous good books discuss UML; for example [105, 278, 640] and there is a yearly conference dedicated to the advancement of UML, the International Conference on the Unified Modeling Language.

View integration is discussed in several papers, including [97, 139, 184, 244, 535, 551, 550, 685, 697, 748]. [64] is a survey of several integration approaches.



3

THE RELATIONAL MODEL

- How is data represented in the relational model?
- What integrity constraints can be expressed?
- How can data be created and modified?
- How can data be manipulated and queried?
- How can we create, modify, and query tables using SQL?
- How do we obtain a relational database design from an ER diagram?
- What are views and why are they used?
- **Key concepts:** relation, schema, instance, tuple, field, domain, degree, cardinality; SQL DDL, `CREATE TABLE`, `INSERT`, `DELETE`, `UPDATE`; integrity constraints, domain constraints, key constraints, `PRIMARY KEY`, `UNIQUE`, foreign key constraints, `FOREIGN KEY`; referential integrity maintenance, deferred and immediate constraints; relational queries; logical database design, translating ER diagrams to relations, expressing ER constraints using SQL; views, views and logical independence, security; creating views in SQL, updating views, querying views, dropping views

TABLE: An arrangement of words, numbers, or signs, or combinations of them, as in parallel columns, to exhibit a set of facts or relations in a definite, compact, and comprehensive form; a synopsis or scheme.

—Webster's *Dictionary of the English Language*

Codd proposed the relational data model in 1970. At that time, most database systems were based on one of two older data models (the hierarchical model

SQL. Originally developed as the query language of the pioneering System-R relational DBMS at IBM, structured query language (SQL) has become the most widely used language for creating, manipulating, and querying relational DBMSs. Since many vendors offer SQL products, there is a need for a standard that defines 'official SQL.' The existence of a standard allows users to measure a given vendor's version of SQL for completeness. It also allows users to distinguish SQL features specific to one product from those that are standard; an application that relies on nonstandard features is less portable.

The first SQL standard was developed in 1986 by the American National Standards Institute (ANSI) and was called SQL-86. There was a minor revision in 1989 called SQL-89 and a major revision in 1992 called SQL-92. The International Standards Organization (ISO) collaborated with ANSI to develop SQL-92. Most commercial DBMSs currently support (the core subset of) SQL-92 and are working to support the recently adopted SQL:1999 version of the standard, a major extension of SQL-92. Our coverage of SQL is based on SQL:1999, but is applicable to SQL-92 as well; features unique to SQL:1999 are explicitly noted.

and the network model); the relational model revolutionized the database field and largely supplanted these earlier models. Prototype relational database management systems were developed in pioneering research projects at IBM and UC-Berkeley by the mid-1970s, and several vendors were offering relational database products shortly thereafter. Today, the relational model is by far the dominant data model and the foundation for the leading DBMS products, including IBM's DB2 family, Informix, Oracle, Sybase, Microsoft's Access and SQLServer, FoxBase, and Paradox. Relational database systems are ubiquitous in the marketplace and represent a multibillion dollar industry.

The relational model is very simple and elegant: a database is a collection of one or more *relations*, where each relation is a table with rows and columns. This simple tabular representation enables even novice users to understand the contents of a database, and it permits the use of simple, high-level languages to query the data. The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

While we concentrate on the underlying concepts, we also introduce the **Data Definition Language (DDL)** features of SQL, the standard language for creating, manipulating, and querying data in a relational DBMS. This allows us to ground the discussion firmly in terms of real database systems.

We discuss the concept of a relation in Section 3.1 and show how to create relations using the SQL language. An important component of a data model is the set of constructs it provides for specifying conditions that must be satisfied by the data. Such conditions, called *integrity constraints* (ICs), enable the DBMS to reject operations that might corrupt the data. We present integrity constraints in the relational model in Section 3.2, along with a discussion of SQL support for ICs. We discuss how a DBMS enforces integrity constraints in Section 3.3.

In Section 3.4, we turn to the mechanism for accessing and retrieving data from the database, *query languages*, and introduce the querying features of SQL, which we examine in greater detail in a later chapter.

We then discuss converting an ER diagram into a relational database schema in Section 3.5. We introduce *views*, or tables defined using queries, in Section 3.6. Views can be used to define the external schema for a database and thus provide the support for logical data independence in the relational model. In Section 3.7, we describe SQL commands to destroy and alter tables and views.

Finally, in Section 3.8 we extend our design case study, the Internet shop introduced in Section 2.8, by showing how the ER diagram for its conceptual schema can be mapped to the relational model, and how the use of views can help in this design.

3.1 INTRODUCTION TO THE RELATIONAL MODEL

The main construct for representing data in the relational model is a **relation**. A relation consists of a **relation schema** and a **relation instance**. The relation instance is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance. The schema specifies the relation's name, the name of each **field** (or **column**, or **attribute**), and the **domain** of each field. A domain is referred to in a relation schema by the **domain name** and has a set of associated **values**.

We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)

This says, for instance, that the field named *sid* has a domain named `string`. The set of values associated with domain `string` is the set of all character strings.

We now turn to the instances of a relation. An **instance** of a relation is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a *table* in which each tuple is a *row*, and all rows have the same number of fields. (The term *relation instance* is often abbreviated to just *relation*, when there is no confusion with other aspects of a relation such as its schema.)

An instance of the Students relation appears in Figure 3.1. The instance $S1$

Field names

FIELDS (ATTRIBUTES, COLUMNS)

TUPLES (RECORDS, ROWS)

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance $S1$ of the Students Relation

contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model—each relation is defined to be a *set* of unique tuples or rows.

In practice, commercial systems allow tables to have duplicate rows, but we assume that a relation is indeed a set of tuples unless otherwise noted. The order in which the rows are listed is not important. Figure 3.2 shows the same relation instance. If the fields are named, as in our schema definitions and

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53666	Jones	jones@cs	18	3.4
50000	Dave	dave@cs	19	3.3

Figure 3.2 An Alternative Representation of Instance $S1$ of Students

figures depicting relation instances, the order of fields does not matter either. However, an alternative convention is to list fields in a specific order and refer

to a field by its position. Thus, *sid* is field 1 of Students, *login* is field 3, and so on. If this convention is used, the order of fields is significant. Most database systems use a combination of these conventions. For example, in SQL, the named fields convention is used in statements that retrieve tuples and the ordered fields convention is commonly used when inserting tuples.

A relation schema specifies the domain of each field or column in the relation instance. These **domain constraints** in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the *type* of that field, in programming language terms, and restricts the values that can appear in the field.

More formally, let R(f_1:D1, ..., f_n:Dn) be a relation schema, and for each f_i, $1 \leq i \leq n$, let Dom_i be the set of values associated with the domain named Di. An instance of R that satisfies the domain constraints in the schema is a set of tuples with n fields:

$$\{ \langle f_1 : d_1, \ \ldots \ , f_n : d_n \rangle \ \mid \ d_1 \in Dom_1, \ \ldots \ , d_n \in Dom_n \ \}$$

The angular brackets $\langle \ldots \rangle$ identify the fields of a tuple. Using this notation, the first Students tuple shown in Figure 3.1 is written as $\langle$*sid:* 50000, *name:* Dave, *login:* dave@cs, *age:* 19, *gpa:* 3.3$\rangle$. The curly brackets $\{\ldots\}$ denote a set (of tuples, in this definition). The vertical bar | should be read 'such that,' the symbol $\in$ should be read 'in,' and the expression to the right of the vertical bar is a condition that must be satisfied by the field values of each tuple in the set. Therefore, an instance of R is defined as a set of tuples. The fields of each tuple must correspond to the fields in the relation schema.

Domain constraints are so fundamental in the relational model that we henceforth consider only relation instances that satisfy them; therefore, *relation instance* means *relation instance that satisfies the domain constraints in the relation schema.*

The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality** of a relation instance is the number of tuples in it. In Figure 3.1, the degree of the relation (the number of columns) is five, and the cardinality of this instance is six.

A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database. For example, in Chapter 1, we discussed a university database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In. An **instance** of a relational database is a collection of relation

instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

3.1.1 Creating and Modifying Relations Using SQL

The SQL language standard uses the word *table* to denote *relation*, and we often follow this convention when discussing SQL. The subset of SQL that supports the creation, deletion, and modification of tables is called the Data Definition Language (DDL). Further, while there is a command that lets users define new domains, analogous to type definition commands in a programming language, we postpone a discussion of domain definition until Section 5.7. For now, we only consider domains that are built-in types, such as `integer`.

The `CREATE TABLE` statement is used to define a new table.[1] To create the Students relation, we can use the following statement:

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        gpa    REAL )
```

Tuples are inserted using the `INSERT` command. We can insert a single tuple into the Students table as follows:

```
INSERT
INTO   Students   (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

We can optionally omit the list of column names in the `INTO` clause and list the values in the appropriate order, but it is good style to be explicit about column names.

We can delete tuples using the `DELETE` command. We can delete all Students tuples with *name* equal to Smith using the command:

```
DELETE
FROM   Students S
WHERE  S.name = 'Smith'
```

[1] SQL also provides statements to destroy tables and to change the columns associated with a table; we discuss these in Section 3.7.

We can modify the column values in an existing row using the UPDATE command. For example, we can increment the age and decrement the gpa of the student with *sid* 53688:

```
UPDATE Students S
SET    S.age = S.age + 1, S.gpa = S.gpa − 1
WHERE  S.sid = 53688
```

These examples illustrate some important points. The WHERE clause is applied first and determines which rows are to be modified. The SET clause then determines how these rows are to be modified. If the column being modified is also used to determine the new value, the value used in the expression on the right side of equals (=) is the *old* value, that is, before the modification. To illustrate these points further, consider the following variation of the previous query:

```
UPDATE Students S
SET    S.gpa = S.gpa − 0.1
WHERE  S.gpa >= 3.3
```

If this query is applied on the instance $S1$ of Students shown in Figure 3.1, we obtain the instance shown in Figure 3.3.

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.2
53666	Jones	jones@cs	18	3.3
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.7
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.3 Students Instance $S1$ after Update

3.2 INTEGRITY CONSTRAINTS OVER RELATIONS

A database is only as good as the information stored in it, and a DBMS must therefore help prevent the entry of incorrect information. An **integrity constraint (IC)** is a condition specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance. A DBMS **enforces** integrity constraints, in that it permits only legal instances to be stored in the database.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.

2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. (In some situations, rather than disallow the change, the DBMS might make some compensating changes to the data to ensure that the database instance satisfies all ICs. In any case, changes to the database are not allowed to create an instance that violates any IC.) It is important to specify exactly when integrity constraints are checked relative to the statement that causes the change in the data and the transaction that it is part of. We discuss this aspect in Chapter 16, after presenting the transaction concept, which we introduced in Chapter 1, in more detail.

Many kinds of integrity constraints can be specified in the relational model. We have already seen one example of an integrity constraint in the *domain constraints* associated with a relation schema (Section 3.1). In general, other kinds of constraints can be specified as well; for example, no two students have the same *sid* value. In this section we discuss the integrity constraints, other than domain constraints, that a DBA or user can specify in the relational model.

3.2.1 Key Constraints

Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a key constraint. A **key constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple. A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just *key*. In the case of the Students relation, the (set of fields containing just the) *sid* field is a candidate key.

Let us take a closer look at the above definition of a (candidate) key. There are two parts to the definition:[2]

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.

2. No subset of the set of fields in a key is a unique identifier for a tuple.

[2]The term *key* is rather overworked. In the context of access methods, we speak of *search keys*, which are quite different.

The first part of the definition means that, in *any* legal instance, the values in the key fields uniquely identify a tuple in the instance. When specifying a key constraint, the DBA or user must be sure that this constraint will not prevent them from storing a 'correct' set of tuples. (A similar comment applies to the specification of other kinds of ICs as well.) The notion of 'correctness' here depends on the nature of the data being stored. For example, several students may have the same name, although each student has a unique student id. If the *name* field is declared to be a key, the DBMS will not allow the Students relation to contain two tuples describing different students with the same name!

The second part of the definition means, for example, that the set of fields {*sid, name*} is not a key for Students, because this set properly contains the key {*sid*}. The set {*sid, name*} is an example of a **superkey**, which is a set of fields that contains a key.

Look again at the instance of the Students relation in Figure 3.1. Observe that two different rows always have different *sid* values; *sid* is a key and uniquely identifies a tuple. However, this does not hold for nonkey fields. For example, the relation contains two rows with *Smith* in the *name* field.

Note that every relation is guaranteed to have a key. Since a relation is a set of tuples, the set of *all* fields is always a superkey. If other constraints hold, some subset of the fields may form a key, but if not, the set of all fields is a key.

A relation may have several candidate keys. For example, the *login* and *age* fields of the Students relation may, taken together, also identify students uniquely. That is, {*login, age*} is also a key. It may seem that *login* is a key, since no two rows in the example instance have the same *login* value. However, the key must identify tuples uniquely in all possible legal instances of the relation. By stating that {*login, age*} is a key, the user is declaring that two students may have the same login or age, but not both.

Out of all the available candidate keys, a database designer can identify a **primary** key. Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its *sid* value. As a consequence of referring to student tuples in this manner, tuples are frequently accessed by specifying their *sid* value. In principle, we can use any key, not just the primary key, to refer to a tuple. However, using the primary key is preferable because it is what the DBMS expects—this is the significance of designating a particular candidate key as a primary key—and optimizes for. For example, the DBMS may create an index with the primary key fields as the search key, to make the retrieval of a tuple given its primary key value efficient. The idea of referring to a tuple is developed further in the next section.

Specifying Key Constraints in SQL

In SQL, we can declare that a subset of the columns of a table constitute a key by using the UNIQUE constraint. At most one of these candidate keys can be declared to be a *primary key*, using the PRIMARY KEY constraint. (SQL does not require that such constraints be declared for a table.)

Let us revisit our example table definition and specify key information:

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        gpa    REAL,
                        UNIQUE (name, age),
                        CONSTRAINT StudentsKey PRIMARY KEY (sid) )
```

This definition says that *sid* is the primary key and the combination of *name* and *age* is also a key. The definition of the primary key also illustrates how we can name a constraint by preceding it with CONSTRAINT *constraint-name*. If the constraint is violated, the constraint name is returned and can be used to identify the error.

3.2.2 Foreign Key Constraints

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common IC involving two relations is a *foreign key* constraint.

Suppose that, in addition to Students, we have a second relation:

Enrolled(*studid:* `string`, *cid:* `string`, *grade:* `string`)

To ensure that only bona fide students can enroll in courses, any value that appears in the *studid* field of an instance of the Enrolled relation should also appear in the *sid* field of some tuple in the Students relation. The *studid* field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students); that is, it must have the same number of columns and compatible data types, although the column names can be different.

This constraint is illustrated in Figure 3.4. As the figure shows, there may well be some Students tuples that are not referenced from Enrolled (e.g., the student with *sid=50000*). However, every *studid* value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

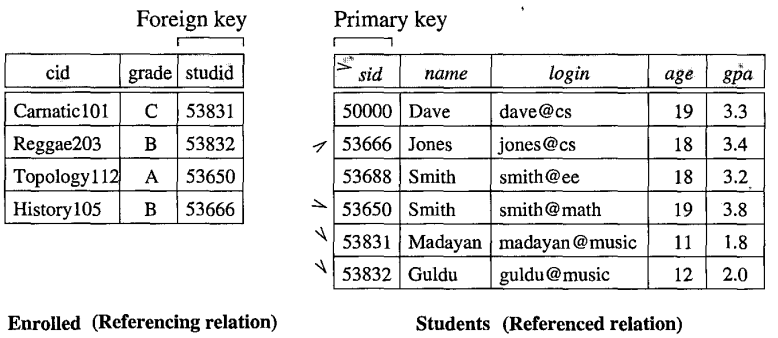


Figure 3.4 Referential Integrity

If we try to insert the tuple ⟨*55555, Art104, A*⟩ into $E1$, the IC is violated because there is no tuple in $S1$ with *sid* 55555; the database system should reject such an insertion. Similarly, if we delete the tuple ⟨*53666, Jones, jones@cs, 18, 3.4*⟩ from $S1$, we violate the foreign key constraint because the tuple ⟨*53666, History105, B*⟩ in $E1$ contains *studid* value 53666, the *sid* of the deleted Students tuple. The DBMS should disallow the deletion or, perhaps, also delete the Enrolled tuple that refers to the deleted Students tuple. We discuss foreign key constraints and their impact on updates in Section 3.3.

Finally, we note that a foreign key could refer to the same relation. For example, we could extend the Students relation with a column called *partner* and declare this column to be a foreign key referring to Students. Intuitively, every student could then have a partner, and the *partner* field contains the partner's *sid*. The observant reader will no doubt ask, "What if a student does not (yet) have a partner?" This situation is handled in SQL by using a special value called **null**. The use of *null* in a field of a tuple means that value in that field is either unknown or not applicable (e.g., we do not know the partner yet or there is no partner). The appearance of *null* in a foreign key field does not violate the foreign key constraint. However, *null* values are not allowed to appear in a primary key field (because the primary key fields are used to identify a tuple uniquely). We discuss *null* values further in Chapter 5.

Specifying Foreign Key Constraints in SQL

Let us define Enrolled(*studid:* `string`, *cid:* `string`, *grade:* `string`):

```
CREATE TABLE Enrolled ( studid CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (studid, cid),
                        FOREIGN KEY (studid) REFERENCES Students )
```

The foreign key constraint states that every *studid* value in Enrolled must also appear in Students, that is, *studid* in Enrolled is a foreign key referencing Students. Specifically, every *studid* value in Enrolled must appear as the value in the primary key field, *sid*, of Students. Incidentally, the primary key constraint for Enrolled states that a student has exactly one grade for each course he or she is enrolled in. If we want to record more than one grade per student per course, we should change the primary key constraint.

3.2.3 General Constraints

Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.

For example, we may require that student ages be within a certain range of values; given such an IC specification, the DBMS rejects inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, the instance of Students shown in Figure 3.1 is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in Figure 3.5.

sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.5 An Instance $S2$ of the Students Relation

The IC that students must be older than 16 can be thought of as an extended domain constraint, since we are essentially defining the set of permissible *age*

values more stringently than is possible by simply using a standard domain such as `integer`. In general, however, constraints that go well beyond domain, key, or foreign key constraints can be specified. For example, we could require that every student whose age is greater than 18 must have a gpa greater than 3.

Current relational database systems support such general constraints in the form of *table constraints* and *assertions*. Table constraints are associated with a single table and checked whenever that table is modified. In contrast, assertions involve several tables and are checked whenever any of these tables is modified. Both table constraints and assertions can use the full power of SQL queries to specify the desired restriction. We discuss SQL support for *table constraints* and *assertions* in Section 5.7 because a full appreciation of their power requires a good grasp of SQL's query capabilities.

3.3 ENFORCING INTEGRITY CONSTRAINTS

As we observed earlier, ICs are specified when a relation is created and enforced when a relation is modified. The impact of domain, `PRIMARY KEY`, and `UNIQUE` constraints is straightforward: If an insert, delete, or update command causes a violation, it is rejected. Every potential IC violation is generally checked at the end of each SQL statement execution, although it can be *deferred* until the end of the transaction executing the statement, as we will see in Section 3.3.1.

Consider the instance $S1$ of Students shown in Figure 3.1. The following insertion violates the primary key constraint because there is already a tuple with the *sid* 53688, and it will be rejected by the DBMS:

```
INSERT
INTO   Students   (sid, name, login, age, gpa)
VALUES (53688, 'Mike', 'mike@ee', 17, 3.4)
```

The following insertion violates the constraint that the primary key cannot contain *null*:

```
INSERT
INTO   Students   (sid, name, login, age, gpa)
VALUES (null, 'Mike', 'mike@ee', 17, 3.4)
```

Of course, a similar problem arises whenever we try to insert a tuple with a value in a field that is not in the domain associated with that field, that is, whenever we violate a domain constraint. Deletion does not cause a violation of domain, primary key or unique constraints. However, an update can cause violations, similar to an insertion:

```
UPDATE Students S
SET    S.sid = 50000
WHERE  S.sid = 53688
```

This update violates the primary key constraint because there is already a tuple with *sid* 50000.

The impact of foreign key constraints is more complex because SQL sometimes tries to rectify a foreign key constraint violation instead of simply rejecting the change. We discuss the **referential integrity enforcement steps** taken by the DBMS in terms of our Enrolled and Students tables, with the foreign key constraint that Enrolled.*sid* is a reference to (the primary key of) Students.

In addition to the instance $S1$ of Students, consider the instance of Enrolled shown in Figure 3.4. Deletions of Enrolled tuples do not violate referential integrity, but insertions of Enrolled tuples could. The following insertion is illegal because there is no Students tuple with *sid* 51111:

```
INSERT
INTO   Enrolled   (cid, grade, studid)
VALUES ('Hindi101', 'B', 51111)
```

On the other hand, insertions of Students tuples do not violate referential integrity, and deletions of Students tuples could cause violations. Further, updates on either Enrolled or Students that change the *studid* (respectively, *sid*) value could potentially violate referential integrity.

SQL provides several alternative ways to handle foreign key violations. We must consider three basic questions:

1. *What should we do if an Enrolled row is inserted, with a* studid *column value that does not appear in any row of the Students table?*

   In this case, the `INSERT` command is simply rejected.

2. *What should we do if a Students row is deleted?*

   The options are:

   - Delete all Enrolled rows that refer to the deleted Students row.
   - Disallow the deletion of the Students row if an Enrolled row refers to it.
   - Set the *studid* column to the *sid* of some (existing) 'default' student, for every Enrolled row that refers to the deleted Students row.

- For every Enrolled row that refers to it, set the *studid* column to *null*. In our example, this option conflicts with the fact that *studid* is part of the primary key of Enrolled and therefore cannot be set to *null*. Therefore, we are limited to the first three options in our example, although this fourth option (setting the foreign key to *null*) is available in general.

3. *What should we do if the primary key value of a Students row is updated?* The options here are similar to the previous case.

SQL allows us to choose any of the four options on DELETE and UPDATE. For example, we can specify that when a Students row is *deleted*, all Enrolled rows that refer to it are to be deleted as well, but that when the *sid* column of a Students row is *modified*, this update is to be rejected if an Enrolled row refers to the modified Students row:

```
CREATE TABLE Enrolled ( studid CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (studid, cid),
                        FOREIGN KEY (studid) REFERENCES Students
                                    ON DELETE CASCADE
                                    ON UPDATE NO ACTION )
```

The options are specified as part of the foreign key declaration. The default option is NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected. Thus, the ON UPDATE clause in our example could be omitted, with the same effect. The CASCADE keyword says that, if a Students row is deleted, all Enrolled rows that refer to it are to be deleted as well. If the UPDATE clause specified CASCADE, and the *sid* column of a Students row is updated, this update is also carried out in each Enrolled row that refers to the updated Students row.

If a Students row is deleted, we can switch the enrollment to a 'default' student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the *sid* field in Enrolled; for example, *sid* CHAR(20) DEFAULT *'53666'*. Although the specification of a default value is appropriate in some situations (e.g., a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE) or to reject the update.

SQL also allows the use of *null* as the default value by specifying ON DELETE SET NULL.

3.3.1 Transactions and Constraints

As we saw in Chapter 1, a program that runs against a database is called a transaction, and it can contain several statements (queries, inserts, updates, etc.) that access the database. If (the execution of) a statement in a transaction violates an integrity constraint, should the DBMS detect this right away or should all constraints be checked together just before the transaction completes?

By default, a constraint is checked at the end of every SQL statement that could lead to a violation, and if there is a violation, the statement is rejected. Sometimes this approach is too inflexible. Consider the following variants of the Students and Courses relations; every student is required to have an honors course, and every course is required to have a grader, who is some student.

```
CREATE TABLE Students ( sid     CHAR(20),
                        name    CHAR(30),
                        login   CHAR(20),
                        age     INTEGER,
                        honors  CHAR(10) NOT NULL,
                        gpa     REAL )
                        PRIMARY KEY (sid),
                        FOREIGN KEY (honors) REFERENCES Courses (cid))

CREATE TABLE Courses (  cid     CHAR(10),
                        cname   CHAR(10),
                        credits INTEGER,
                        grader  CHAR(20) NOT NULL,
                        PRIMARY KEY (cid)
                        FOREIGN KEY (grader) REFERENCES Students (sid))
```

Whenever a Students tuple is inserted, a check is made to see if the honors course is in the Courses relation, and whenever a Courses tuple is inserted, a check is made to see that the grader is in the Students relation. How are we to insert the very first course or student tuple? One cannot be inserted without the other. The only way to accomplish this insertion is to **defer** the constraint checking that would normally be carried out at the end of an INSERT statement.

SQL allows a constraint to be in DEFERRED or IMMEDIATE mode.

```
SET CONSTRAINT ConstraintFoo DEFERRED
```

A constraint in deferred mode is checked at commit time. In our example, the foreign key constraints on Boats and Sailors can both be declared to be in deferred mode. We can then insert a boat with a nonexistent sailor as the captain (temporarily making the database inconsistent), insert the sailor (restoring consistency), then commit and check that both constraints are satisfied.

3.4 QUERYING RELATIONAL DATA

A **relational database query** (query, for short) is a question about the data, and the answer consists of a new relation containing the result. For example, we might want to find all students younger than 18 or all students enrolled in Reggae203. A **query language** is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. We now present some SQL examples that illustrate how easily relations can be queried. Consider the instance of the Students relation shown in Figure 3.1. We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

```
SELECT *
FROM   Students S
WHERE  S.age < 18
```

The symbol '*' means that we retain all fields of selected tuples in the result. Think of S as a variable that takes on the value of each tuple in Students, one tuple after the other. The condition *S.age < 18* in the `WHERE` clause specifies that we want to select only tuples in which the *age* field has a value less than 18. This query evaluates to the relation shown in Figure 3.6.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.6 Students with *age* < 18 on Instance *S1*

This example illustrates that the domain of a field restricts the operations that are permitted on field values, in addition to restricting the values that can appear in the field. The condition *S.age < 18* involves an arithmetic comparison of an *age* value with an integer and is permissible because the domain of *age* is the set of integers. On the other hand, a condition such as *S.age = S.sid* does not make sense because it compares an integer value with a string value, and this comparison is defined to fail in SQL; a query containing this condition produces no answer tuples.

In addition to selecting a subset of tuples, a query can extract a subset of the fields of each selected tuple. We can compute the names and logins of students who are younger than 18 with the following query:

```
SELECT S.name, S.login
FROM   Students S
WHERE  S.age < 18
```

Figure 3.7 shows the answer to this query; it is obtained by applying the selection to the instance $S1$ of Students (to get the relation shown in Figure 3.6), followed by removing unwanted fields. Note that the order in which we perform these operations does matter—if we remove unwanted fields first, we cannot check the condition $S.age < 18$, which involves one of those fields.

name	*login*
Madayan	madayan@music
Guldu	guldu@music

Figure 3.7 Names and Logins of Students under 18

We can also combine information in the Students and Enrolled relations. If we want to obtain the names of all students who obtained an A and the id of the course in which they got an A, we could write the following query:

```
SELECT S.name, E.cid
FROM   Students S, Enrolled E
WHERE  S.sid = E.studid AND E.grade = 'A'
```

This query can be understood as follows: "If there is a Students tuple S and an Enrolled tuple E such that S.sid = E.studid (so that S describes the student who is enrolled in E) and E.grade = 'A', then print the student's name and the course id." When evaluated on the instances of Students and Enrolled in Figure 3.4, this query returns a single tuple, $\langle Smith, Topology112 \rangle$.

We cover relational queries and SQL in more detail in subsequent chapters.

3.5 LOGICAL DATABASE DESIGN: ER TO RELATIONAL

The ER model is convenient for representing an initial, high-level database design. Given an ER diagram describing a database, a standard approach is taken to generating a relational database schema that closely approximates

the ER design. (The translation is approximate to the extent that we cannot capture all the constraints implicit in the ER design using SQL, unless we use certain SQL constraints that are costly to check.) We now describe how to translate an ER diagram into a collection of tables with associated constraints, that is, a relational database schema.

3.5.1 Entity Sets to Tables

An entity set is mapped to a relation in a straightforward way: Each attribute of the entity set becomes an attribute of the table. Note that we know both the domain of each attribute and the (primary) key of an entity set.

Consider the Employees entity set with attributes *ssn, name,* and *lot* shown in Figure 3.8. A possible instance of the Employees entity set, containing three

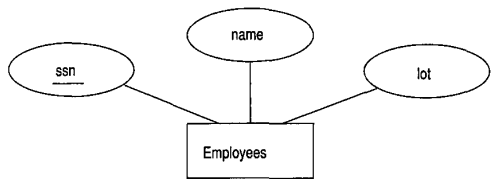


Figure 3.8 The Employees Entity Set

Employees entities, is shown in Figure 3.9 in a tabular format.

ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

Figure 3.9 An Instance of the Employees Entity Set

The following SQL statement captures the preceding information, including the domain constraints and key information:

```
CREATE TABLE Employees ( ssn      CHAR(11),
                         name     CHAR(30),
                         lot      INTEGER,
                         PRIMARY KEY (ssn) )
```

3.5.2 Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model. We begin by considering relationship sets without key and participation constraints, and we discuss how to handle such constraints in subsequent sections. To represent a relationship, we must be able to identify each participating entity and give values to the descriptive attributes of the relationship. Thus, the attributes of the relation include:

- The primary key attributes of each participating entity set, as foreign key fields.
- The descriptive attributes of the relationship set.

The set of nondescriptive attributes is a superkey for the relation. If there are no key constraints (see Section 2.4.1), this set of attributes is a candidate key.

Consider the Works_In2 relationship set shown in Figure 3.10. Each department has offices in several locations and we want to record the locations at which each employee works.

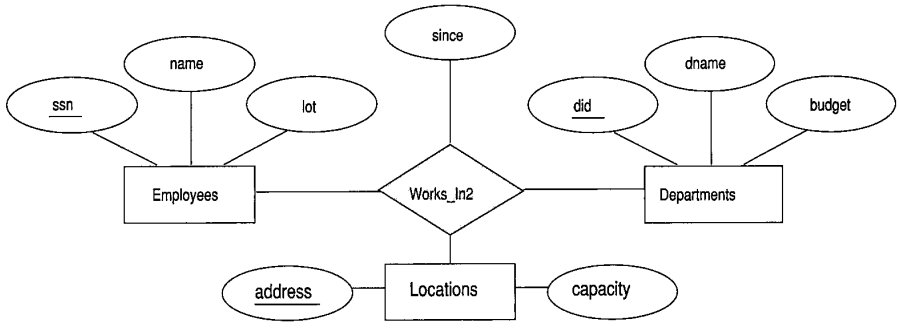


Figure 3.10 A Ternary Relationship Set

All the available information about the Works_In2 table is captured by the following SQL definition:

```
CREATE TABLE Works_In2 ( ssn      CHAR(11),
                         did      INTEGER,
                         address  CHAR(20),
                         since    DATE,
                         PRIMARY KEY (ssn, did, address),
                         FOREIGN KEY (ssn) REFERENCES Employees,
```

```
                    FOREIGN KEY (address) REFERENCES Locations,
                    FOREIGN KEY (did) REFERENCES Departments )
```

Note that the *address*, *did*, and *ssn* fields cannot take on *null* values. Because these fields are part of the primary key for Works_In2, a NOT NULL constraint is implicit for each of these fields. This constraint ensures that these fields uniquely identify a department, an employee, and a location in each tuple of Works_In. We can also specify that a particular action is desired when a referenced Employees, Departments, or Locations tuple is deleted, as explained in the discussion of integrity constraints in Section 3.2. In this chapter, we assume that the default action is appropriate except for situations in which the semantics of the ER diagram require some other action.

Finally, consider the Reports_To relationship set shown in Figure 3.11. The

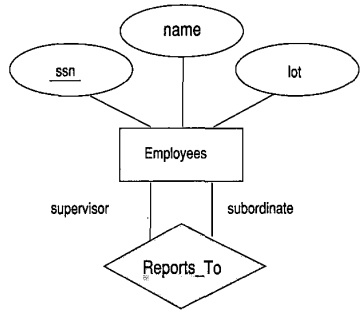


Figure 3.11 The Reports_To Relationship Set

role indicators *supervisor* and *subordinate* are used to create meaningful field names in the CREATE statement for the Reports_To table:

```
CREATE TABLE Reports_To (
        supervisor_ssn    CHAR(11),
        subordinate_ssn   CHAR(11),
        PRIMARY KEY (supervisor_ssn, subordinate_ssn),
        FOREIGN KEY (supervisor_ssn) REFERENCES Employees(ssn),
        FOREIGN KEY (subordinate_ssn) REFERENCES Employees(ssn) )
```

Observe that we need to explicitly name the referenced field of Employees because the field name differs from the name(s) of the referring field(s).

3.5.3 Translating Relationship Sets with Key Constraints

If a relationship set involves n entity sets and some m of them are linked via arrows in the ER diagram, the key for any one of these m entity sets constitutes a key for the relation to which the relationship set is mapped. Hence we have m candidate keys, and one of these should be designated as the primary key. The translation discussed in Section 2.3 from relationship sets to a relation can be used in the presence of key constraints, taking into account this point about keys.

Consider the relationship set Manages shown in Figure 3.12. The table cor-

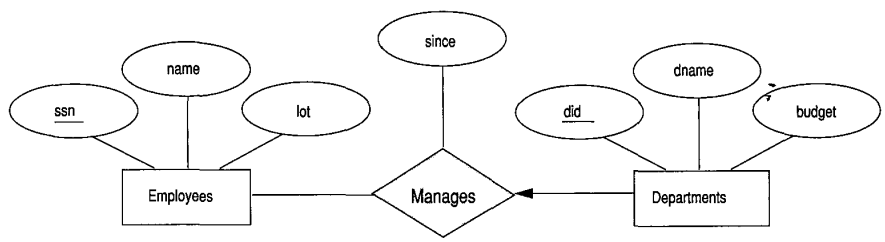


Figure 3.12 Key Constraint on Manages

responding to Manages has the attributes *ssn, did, since.* However, because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value. A consequence of this observation is that *did* is itself a key for Manages; indeed, the set *did, ssn* is not a key (because it is not minimal). The Manages relation can be defined using the following SQL statement:

```
CREATE TABLE Manages ( ssn      CHAR(11),
                       did      INTEGER,
                       since    DATE,
                       PRIMARY KEY (did),
                       FOREIGN KEY (ssn) REFERENCES Employees,
                       FOREIGN KEY (did) REFERENCES Departments )
```

A second approach to translating a relationship set with key constraints is often superior because it avoids creating a distinct table for the relationship set. The idea is to include the information about the relationship set in the table corresponding to the entity set with the key, taking advantage of the key constraint. In the Manages example, because a department has at most one manager, we can add the key fields of the Employees tuple denoting the manager and the *since* attribute to the Departments tuple.

This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations. The only drawback to this approach is that space could be wasted if several departments have no managers. In this case the added fields would have to be filled with *null* values. The first translation (using a separate table for Manages) avoids this inefficiency, but some important queries require us to combine information from two relations, which can be a slow operation.

The following SQL statement, defining a Dept_Mgr relation that captures the information in both Departments and Manages, illustrates the second approach to translating relationship sets with key constraints:

```
CREATE TABLE Dept_Mgr ( did      INTEGER,
                        dname    CHAR(20),
                        budget   REAL,
                        ssn      CHAR(11),
                        since    DATE,
                        PRIMARY KEY (did),
                        FOREIGN KEY (ssn) REFERENCES Employees )
```

Note that *ssn* can take on *null* values.

This idea can be extended to deal with relationship sets involving more than two entity sets. In general, if a relationship set involves n entity sets and some m of them are linked via arrows in the ER diagram, the relation corresponding to any one of the m sets can be augmented to capture the relationship.

We discuss the relative merits of the two translation approaches further after considering how to translate relationship sets with participation constraints into tables.

3.5.4 Translating Relationship Sets with Participation Constraints

Consider the ER diagram in Figure 3.13, which shows two relationship sets, Manages and Works_In.

Every department is required to have a manager, due to the participation constraint, and at most one manager, due to the key constraint. The following SQL statement reflects the second translation approach discussed in Section 3.5.3, and uses the key constraint:

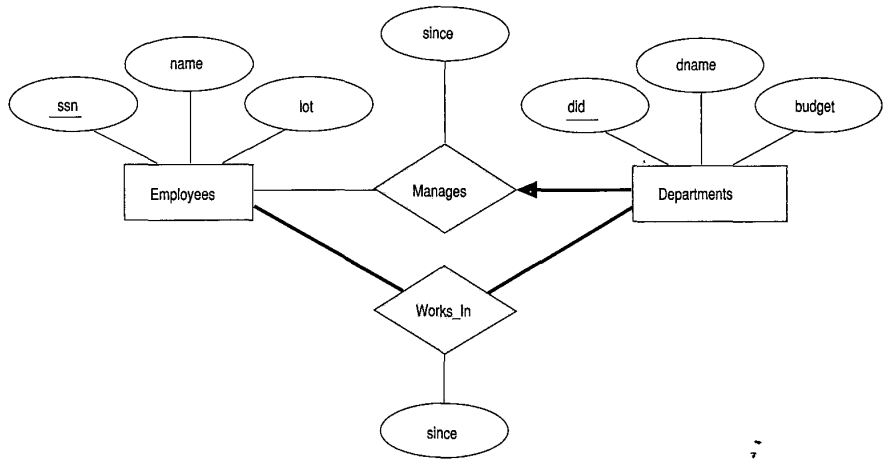


Figure 3.13 Manages and Works_In

```
CREATE TABLE Dept_Mgr ( did      INTEGER,
                        dname    CHAR(20),
                        budget   REAL,
                        ssn      CHAR(11) NOT NULL,
                        since    DATE,
                        PRIMARY KEY (did),
                        FOREIGN KEY (ssn) REFERENCES Employees
                                ON DELETE NO ACTION )
```

It also captures the participation constraint that every department must have a manager: Because *ssn* cannot take on *null* values, each tuple of Dept_Mgr identifies a tuple in Employees (who is the manager). The `NO ACTION` specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while it is pointed to by a Dept_Mgr tuple. If we wish to delete such an Employees tuple, we must first change the Dept_Mgr tuple to have a new employee as manager. (We could have specified `CASCADE` instead of `NO ACTION`, but deleting all information about a department just because its manager has been fired seems a bit extreme!)

The constraint that every department must have a manager cannot be captured using the first translation approach discussed in Section 3.5.3. (Look at the definition of Manages and think about what effect it would have if we added `NOT NULL` constraints to the *ssn* and *did* fields. *Hint:* The constraint would prevent the firing of a manager, but does not ensure that a manager is initially appointed for each department!) This situation is a strong argument

in favor of using the second approach for one-to-many relationships such as Manages, especially when the entity set with the key constraint also has a total participation constraint.

Unfortunately, there are many participation constraints that we cannot capture using SQL, short of using *table constraints* or *assertions*. Table constraints and assertions can be specified using the full power of the SQL query language (as discussed in Section 5.7) and are very expressive but also very expensive to check and enforce. For example, we cannot enforce the participation constraints on the Works_In relation without using these general constraints. To see why, consider the Works_In relation obtained by translating the ER diagram into relations. It contains fields *ssn* and *did*, which are foreign keys referring to Employees and Departments. To ensure total participation of Departments in Works_In, we have to guarantee that every *did* value in Departments appears in a tuple of Works_In. We could try to guarantee this condition by declaring that *did* in Departments is a foreign key referring to Works_In, but this is not a valid foreign key constraint because *did* is not a candidate key for Works_In.

To ensure total participation of Departments in Works_In using SQL, we need an assertion. We have to guarantee that every *did* value in Departments appears in a tuple of Works_In; further, this tuple of Works_In must also have non-*null* values in the fields that are foreign keys referencing other entity sets involved in the relationship (in this example, the *ssn* field). We can ensure the second part of this constraint by imposing the stronger requirement that *ssn* in Works_In cannot contain *null* values. (Ensuring that the participation of Employees in Works_In is total is symmetric.)

Another constraint that requires assertions to express in SQL is the requirement that each Employees entity (in the context of the Manages relationship set) must manage at least one department.

In fact, the Manages relationship set exemplifies most of the participation constraints that we can capture using key and foreign key constraints. Manages is a binary relationship set in which exactly one of the entity sets (Departments) has a key constraint, and the total participation constraint is expressed on that entity set.

We can also capture participation constraints using key and foreign key constraints in one other special situation: a relationship set in which all participating entity sets have key constraints and total participation. The best translation approach in this case is to map all the entities as well as the relationship into a single table; the details are straightforward.

3.5.5 Translating Weak Entity Sets

A weak entity set always participates in a one-to-many binary relationship and has a key constraint and total participation. The second translation approach discussed in Section 3.5.3 is ideal in this case, but we must take into account that the weak entity has only a partial key. Also, when an owner entity is deleted, we want all owned weak entities to be deleted.

Consider the Dependents weak entity set shown in Figure 3.14, with partial key *pname*. A Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity, and the Dependents entity must be deleted if the owning Employees entity is deleted.

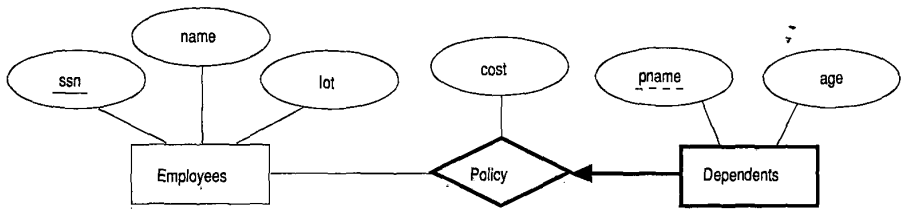


Figure 3.14 The Dependents Weak Entity Set

We can capture the desired semantics with the following definition of the Dep_Policy relation:

```
CREATE TABLE Dep_Policy ( pname   CHAR(20),
                          age     INTEGER,
                          cost    REAL,
                          ssn     CHAR(11),
                          PRIMARY KEY (pname, ssn),
                          FOREIGN KEY (ssn) REFERENCES Employees
                                  ON DELETE CASCADE )
```

Observe that the primary key is ⟨*pname, ssn*⟩, since Dependents is a weak entity. This constraint is a change with respect to the translation discussed in Section 3.5.3. We have to ensure that every Dependents entity is associated with an Employees entity (the owner), as per the total participation constraint on Dependents. That is, *ssn* cannot be *null*. This is ensured because *ssn* is part of the primary key. The CASCADE option ensures that information about an employee's policy and dependents is deleted if the corresponding Employees tuple is deleted.

3.5.6 Translating Class Hierarchies

We present the two basic approaches to handling ISA hierarchies by applying them to the ER diagram shown in Figure 3.15:

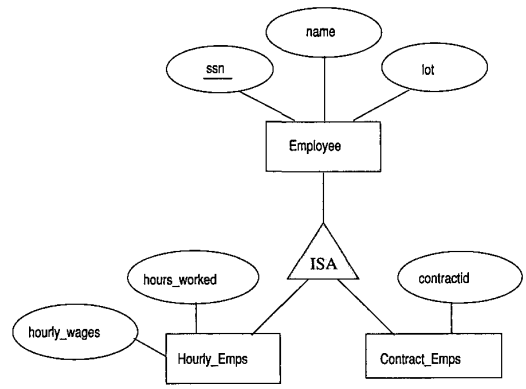


Figure 3.15 Class Hierarchy

1. We can map each of the entity sets Employees, Hourly_Emps, and Contract_Emps to a distinct relation. The Employees relation is created as in Section 2.2. We discuss Hourly_Emps here; Contract_Emps is handled similarly. The relation for Hourly_Emps includes the *hourly_wages* and *hours_worked* attributes of Hourly_Emps. It also contains the key attributes of the superclass (*ssn*, in this example), which serve as the primary key for Hourly_Emps, as well as a foreign key referencing the superclass (Employees). For each Hourly_Emps entity, the value of the *name* and *lot* attributes are stored in the corresponding row of the superclass (Employees). Note that if the superclass tuple is deleted, the delete must be cascaded to Hourly_Emps.

2. Alternatively, we can create just two relations, corresponding to Hourly_Emps and Contract_Emps. The relation for Hourly_Emps includes all the attributes of Hourly_Emps as well as all the attributes of Employees (i.e., *ssn, name, lot, hourly_wages, hours_worked*).

The first approach is general and always applicable. Queries in which we want to examine all employees and do not care about the attributes specific to the subclasses are handled easily using the Employees relation. However, queries in which we want to examine, say, hourly employees, may require us to combine Hourly_Emps (or Contract_Emps, as the case may be) with Employees to retrieve *name* and *lot*.

The second approach is not applicable if we have employees who are neither hourly employees nor contract employees, since there is no way to store such employees. Also, if an employee is both an Hourly_Emps and a Contract_Emps entity, then the *name* and *lot* values are stored twice. This duplication can lead to some of the anomalies that we discuss in Chapter 19. A query that needs to examine all employees must now examine two relations. On the other hand, a query that needs to examine only hourly employees can now do so by examining just one relation. The choice between these approaches clearly depends on the semantics of the data and the frequency of common operations.

In general, overlap and covering constraints can be expressed in SQL only by using assertions.

3.5.7 Translating ER Diagrams with Aggregation

Consider the ER diagram shown in Figure 3.16. The Employees, Projects,

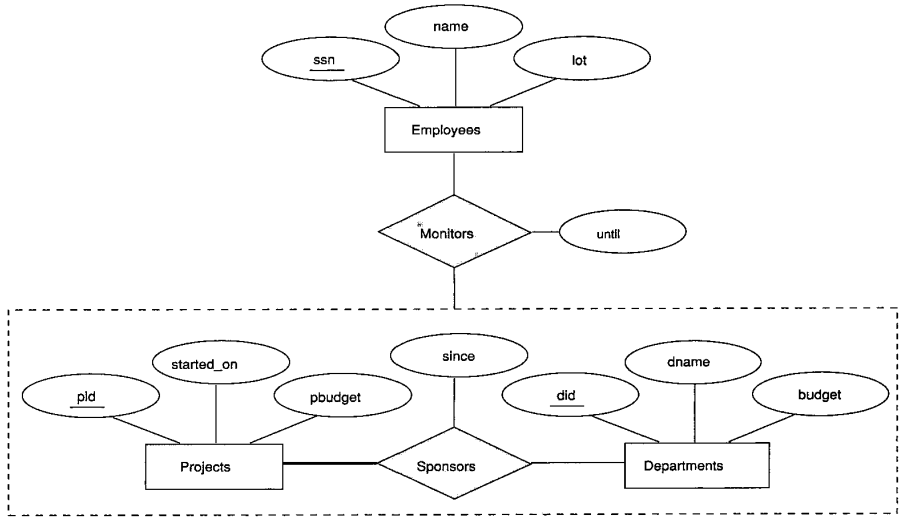


Figure 3.16 Aggregation

and Departments entity sets and the Sponsors relationship set are mapped as described in previous sections. For the Monitors relationship set, we create a relation with the following attributes: the key attributes of Employees (*ssn*), the key attributes of Sponsors (*did, pid*), and the descriptive attributes of Monitors (*until*). This translation is essentially the standard mapping for a relationship set, as described in Section 3.5.2.

There is a special case in which this translation can be refined by dropping the Sponsors relation. Consider the Sponsors relation. It has attributes *pid, did,* and *since*; and in general we need it (in addition to Monitors) for two reasons:

1. We have to record the descriptive attributes (in our example, *since*) of the Sponsors relationship.
2. Not every sponsorship has a monitor, and thus some ⟨*pid, did*⟩ pairs in the Sponsors relation may not appear in the Monitors relation.

However, if Sponsors has no descriptive attributes and has total participation in Monitors, every possible instance of the Sponsors relation can be obtained from the ⟨*pid, did*⟩ columns of Monitors; Sponsors can be dropped.

3.5.8 ER to Relational: Additional Examples

Consider the ER diagram shown in Figure 3.17. We can use the key constraints

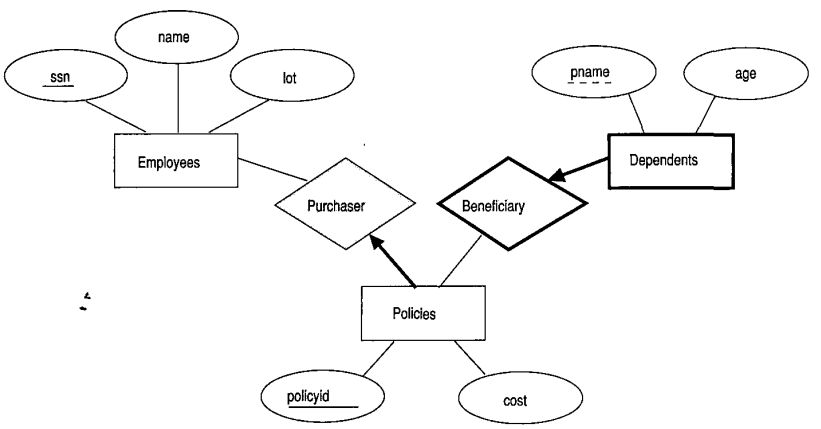


Figure 3.17 Policy Revisited

to combine Purchaser information with Policies and Beneficiary information with Dependents, and translate it into the relational model as follows:

```
CREATE TABLE Policies ( policyid INTEGER,
                        cost     REAL,
                        ssn      CHAR(11) NOT NULL,
                        PRIMARY KEY (policyid),
                        FOREIGN KEY (ssn) REFERENCES Employees
                                ON DELETE CASCADE )
```

```
CREATE TABLE Dependents ( pname   CHAR(20),
                          age     INTEGER,
                          policyid INTEGER,
                          PRIMARY KEY (pname, policyid),
                          FOREIGN KEY (policyid) REFERENCES Policies
                                  ON DELETE CASCADE )
```

Notice how the deletion of an employee leads to the deletion of all policies owned by the employee and all dependents who are beneficiaries of those policies. Further, each dependent is required to have a covering policy—because *policyid* is part of the primary key of Dependents, there is an implicit NOT NULL constraint. This model accurately reflects the participation constraints in the ER diagram and the intended actions when an employee entity is deleted.

In general, there could be a chain of identifying relationships for weak entity sets. For example, we assumed that *policyid* uniquely identifies a policy. Suppose that *policyid* distinguishes only the policies owned by a given employee; that is, *policyid* is only a partial key and Policies should be modeled as a weak entity set. This new assumption about *policyid* does not cause much to change in the preceding discussion. In fact, the only changes are that the primary key of Policies becomes $\langle policyid,\ ssn \rangle$, and as a consequence, the definition of Dependents changes—a field called *ssn* is added and becomes part of both the primary key of Dependents and the foreign key referencing Policies:

```
CREATE TABLE Dependents ( pname   CHAR(20),
                          ssn     CHAR(11),
                          age     INTEGER,
                          policyid INTEGER NOT NULL,
                          PRIMARY KEY (pname, policyid, ssn),
                          FOREIGN KEY (policyid, ssn) REFERENCES Policies
                                  ON DELETE CASCADE )
```

3.6 INTRODUCTION TO VIEWS

A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **view definition**. Consider the Students and Enrolled relations. Suppose we are often interested in finding the names and student identifiers of students who got a grade of B in some course, together with the course identifier. We can define a view for this purpose. Using SQL notation:

```
CREATE VIEW B-Students (name, sid, course)
        AS SELECT S.sname, S.sid, E.cid
```

```
FROM    Students S, Enrolled E
WHERE   S.sid = E.studid AND E.grade = 'B'
```

The view B-Students has three fields called *name*, *sid*, and *course* with the same domains as the fields *sname* and *sid* in *Students* and *cid* in Enrolled. (If the optional arguments *name*, *sid*, and *course* are omitted from the `CREATE VIEW` statement, the column names *sname*, *sid*, and *cid* are inherited.)

This view can be used just like a **base table**, or explicitly stored table, in defining new queries or views. Given the instances of Enrolled and Students shown in Figure 3.4, B-Students contains the tuples shown in Figure 3.18. Conceptually, whenever B-Students is used in a query, the view definition is first evaluated to obtain the corresponding instance of B-Students, then the rest of the query is evaluated treating B-Students like any other relation referred to in the query. (We discuss how queries on views are evaluated in practice in Chapter 25.)

name	*sid*	*course*
Jones	53666	History105
Guldu	53832	Reggae203

Figure 3.18 An Instance of the B-Students View

3.6.1 Views, Data Independence, Security

Consider the levels of abstraction we discussed in Section 1.5.2. The *physical* schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used. The *conceptual* schema is the collection of schemas of the relations stored in the database. While some relations in the conceptual schema can also be exposed to applications, that is, be part of the *external* schema of the database, additional relations in the *external* schema can be defined using the view mechanism. The view mechanism thus provides the support for *logical data independence* in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications. For example, if the schema of a stored relation is changed, we can define a view with the old schema and applications that expect to see the old schema can now use this view.

Views are also valuable in the context of *security*: We can define views that give a group of users access to just the information they are allowed to see. For example, we can define a view that allows students to see the other students'

name and age but not their gpa, and allows all students to access this view but not the underlying Students table (see Chapter 21).

3.6.2 Updates on Views

The motivation behind the view mechanism is to tailor how users see the data. Users should not have to worry about the view versus base table distinction. This goal is indeed achieved in the case of queries on views; a view can be used just like any other relation in defining a query. However, it is natural to want to specify updates on views as well. Here, unfortunately, the distinction between a view and a base table must be kept in mind.

The SQL-92 standard allows updates to be specified only on views that are defined on a single base table using just selection and projection, with no use of aggregate operations.[3] Such views are called **updatable views**. This definition is oversimplified, but it captures the spirit of the restrictions. An update on such a restricted view can always be implemented by updating the underlying base table in an unambiguous way. Consider the following view:

```
CREATE VIEW GoodStudents (sid, gpa)
       AS SELECT S.sid, S.gpa
          FROM   Students S
          WHERE  S.gpa > 3.0
```

We can implement a command to modify the gpa of a GoodStudents row by modifying the corresponding row in Students. We can delete a GoodStudents row by deleting the corresponding row from Students. (In general, if the view did not include a key for the underlying table, several rows in the table could 'correspond' to a single row in the view. This would be the case, for example, if we used *S.sname* instead of *S.sid* in the definition of GoodStudents. A command that affects a row in the view then affects all corresponding rows in the underlying table.)

We can insert a GoodStudents row by inserting a row into Students, using *null* values in columns of Students that do not appear in GoodStudents (e.g., *sname*, *login*). Note that primary key columns are not allowed to contain *null* values. Therefore, if we attempt to insert rows through a view that does not contain the primary key of the underlying table, the insertions will be rejected. For example, if GoodStudents contained *sname* but not *sid*, we could not insert rows into Students through insertions to GoodStudents.

[3]There is also the restriction that the DISTINCT operator cannot be used in updatable view definitions. By default, SQL does not eliminate duplicate copies of rows from the result of a query; the DISTINCT operator requires duplicate elimination. We discuss this point further in Chapter 5.

Updatable Views in SQL:1999 The new SQL standard has expanded the class of view definitions that are updatable, taking primary key constraints into account. In contrast to SQL-92, a view definition that contains more than one table in the `FROM` clause may be updatable under the new definition. Intuitively, we can update a field of a view if it is obtained from exactly one of the underlying tables, and the primary key of that table is included in the fields of the view.

SQL:1999 distinguishes between views whose rows can be modified (*updatable views*) and views into which new rows can be inserted (**insertable-into views**): Views defined using the SQL constructs `UNION`, `INTERSECT`, and `EXCEPT` (which we discuss in Chapter 5) cannot be inserted into, even if they are updatable. Intuitively, updatability ensures that an updated tuple in the view can be traced to exactly one tuple in one of the tables used to define the view. The updatability property, however, may still not enable us to decide into which table to insert a new tuple.

An important observation is that an `INSERT` or `UPDATE` may change the underlying base table so that the resulting (i.e., inserted or modified) row is not in the view! For example, if we try to insert a row $\langle 51234, 2.8 \rangle$ into the view, this row can be (padded with *null* values in the other fields of Students and then) added to the underlying Students table, but it will not appear in the GoodStudents view because it does not satisfy the view condition $gpa > 3.0$. The SQL default action is to allow this insertion, but we can disallow it by adding the clause `WITH CHECK OPTION` to the definition of the view. In this case, only rows that will actually appear in the view are permissible insertions.

We caution the reader, that when a view is defined in terms of another view, the interaction between these view definitions with respect to updates and the `CHECK OPTION` clause can be complex; we not go into the details.

Need to Restrict View Updates

While the SQL rules on updatable views are more stringent than necessary, there are some fundamental problems with updates specified on views and good reason to limit the class of views that can be updated. Consider the Students relation and a new relation called Clubs:

Clubs(*cname:* `string`, *jyear:* `date`, *mname:* `string`)

cname	*jyear*	*mname*
Sailing	1996	Dave
Hiking	1997	Smith
Rowing	1998	Smith

Figure 3.19 An Instance C of Clubs

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.20 An Instance $S3$ of Students

name	*login*	*club*	*since*
Dave	dave@cs	Sailing	1996
Smith	smith@ee	Hiking	1997
Smith	smith@ee	Rowing	1998
Smith	smith@math	Hiking	1997
Smith	smith@math	Rowing	1998

Figure 3.21 Instance of ActiveStudents

A tuple in Clubs denotes that the student called *mname* has been a member of the club *cname* since the date *jyear*.[4] Suppose that we are often interested in finding the names and logins of students with a gpa greater than 3 who belong to at least one club, along with the club name and the date they joined the club. We can define a view for this purpose:

```
CREATE VIEW ActiveStudents (name, login, club, since)
        AS SELECT S.sname, S.login, C.cname, C.jyear
           FROM   Students S, Clubs C
           WHERE  S.sname = C.mname AND S.gpa > 3
```

Consider the instances of Students and Clubs shown in Figures 3.19 and 3.20. When evaluated using the instances C and $S3$, ActiveStudents contains the rows shown in Figure 3.21.

Now suppose that we want to delete the row ⟨*Smith, smith@ee, Hiking, 1997*⟩ from ActiveStudents. How are we to do this? ActiveStudents rows are not stored explicitly but computed as needed from the Students and Clubs tables using the view definition. So we must change either Students or Clubs (or both) in such a way that evaluating the view definition on the modified instance does not produce the row ⟨*Smith, smith@ee, Hiking, 1997.*⟩ This task can be accomplished in one of two ways: by either deleting the row ⟨*53688, Smith, smith@ee, 18, 3.2*⟩ from Students or deleting the row ⟨*Hiking, 1997, Smith*⟩

[4] We remark that Clubs has a poorly designed schema (chosen for the sake of our discussion of view updates), since it identifies students by name, which is not a candidate key for Students.

from Clubs. But neither solution is satisfactory. Removing the Students row has the effect of also deleting the row ⟨*Smith, smith@ee, Rowing, 1998*⟩ from the view ActiveStudents. Removing the Clubs row has the effect of also deleting the row ⟨*Smith, smith@math, Hiking, 1997*⟩ from the view ActiveStudents. Neither side effect is desirable. In fact, the only reasonable solution is to *disallow* such updates on views.

Views involving more than one base table can, in principle, be safely updated. The B-Students view we introduced at the beginning of this section is an example of such a view. Consider the instance of B-Students shown in Figure 3.18 (with, of course, the corresponding instances of Students and Enrolled as in Figure 3.4). To insert a tuple, say ⟨*Dave, 50000, Reggae203*⟩ B-Students, we can simply insert a tuple ⟨*Reggae203, B, 50000*⟩ into Enrolled since there is already a tuple for *sid* 50000 in Students. To insert ⟨*John, 55000, Reggae203*⟩, on the other hand, we have to insert ⟨*Reggae203, B, 55000*⟩ into Enrolled and also insert ⟨*55000, John,* null, null, null⟩ into Students. Observe how *null* values are used in fields of the inserted tuple whose value is not available. Fortunately, the view schema contains the primary key fields of both underlying base tables; otherwise, we would not be able to support insertions into this view. To delete a tuple from the view B-Students, we can simply delete the corresponding tuple from Enrolled.

Although this example illustrates that the SQL rules on updatable views are unnecessarily restrictive, it also brings out the complexity of handling view updates in the general case. For practical reasons, the SQL standard has chosen to allow only updates on a very restricted class of views.

3.7 DESTROYING/ALTERING TABLES AND VIEWS

If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table definition information), we can use the `DROP TABLE` command. For example, `DROP TABLE` Students `RESTRICT` destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword `RESTRICT` is replaced by `CASCADE`, Students is dropped and any referencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the `DROP VIEW` command, which is just like `DROP TABLE`.

`ALTER TABLE` modifies the structure of an existing table. To add a column called *maiden-name* to Students, for example, we would use the following command:

```
ALTER TABLE Students
      ADD COLUMN maiden-name CHAR(10)
```

The definition of Students is modified to add this column, and all existing rows are padded with *null* values in this column. ALTER TABLE can also be used to delete columns and add or drop integrity constraints on a table; we do not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

3.8 CASE STUDY: THE INTERNET STORE

The next design step in our running example, continued from Section 2.8, is logical database design. Using the standard approach discussed in Chapter 3, DBDudes maps the ER diagram shown in Figure 2.20 to the relational model, generating the following tables:

```
CREATE TABLE Books ( isbn             CHAR(10),
                     title            CHAR(80),
                     author           CHAR(80),
                     qty_in_stock     INTEGER,
                     price            REAL,
                     year_published   INTEGER,
                     PRIMARY KEY (isbn))

CREATE TABLE Orders ( isbn          CHAR(10),
                      cid           INTEGER,
                      cardnum       CHAR(16),
                      qty           INTEGER,
                      order_date    DATE,
                      ship_date     DATE,
                      PRIMARY KEY (isbn,cid),
                      FOREIGN KEY (isbn) REFERENCES Books,
                      FOREIGN KEY (cid) REFERENCES Customers )

CREATE TABLE Customers ( cid        INTEGER,
                         cname      CHAR(80),
                         address    CHAR(200),
                         PRIMARY KEY (cid)
```

The design team leader, who is still brooding over the fact that the review exposed a flaw in the design, now has an inspiration. The Orders table contains the field *order_date* and the key for the table contains only the fields *isbn* and *cid*. Because of this, a customer cannot order the same book on different days,

a restriction that was not intended. Why not add the *order_date* attribute to the key for the Orders table? This would eliminate the unwanted restriction:

```
CREATE TABLE Orders (   isbn          CHAR(10),
                        ...
                        PRIMARY KEY (isbn,cid,ship_date),
                        ...)
```

The reviewer, Dude 2, is not entirely happy with this solution, which he calls a 'hack'. He points out that no natural ER diagram reflects this design and stresses the importance of the ER diagram as a design document. Dude 1 argues that, while Dude 2 has a point, it is important to present B&N with a preliminary design and get feedback; everyone agrees with this, and they go back to B&N.

The owner of B&N now brings up some additional requirements he did not mention during the initial discussions: "Customers should be able to purchase several different books in a single order. For example, if a customer wants to purchase three copies of 'The English Teacher' and two copies of 'The Character of Physical Law,' the customer should be able to place a single order for both books."

The design team leader, Dude 1, asks how this affects the shippping policy. Does B&N still want to ship all books in an order together? The owner of B&N explains their shipping policy: "As soon as we have have enough copies of an ordered book we ship it, even if an order contains several books. So it could happen that the three copies of 'The English Teacher' are shipped today because we have five copies in stock, but that 'The Character of Physical Law' is shipped tomorrow, because we currently have only one copy in stock and another copy arrives tomorrow. In addition, my customers could place more than one order per day, and they want to be able to identify the orders they placed."

The DBDudes team thinks this over and identifies two new requirements: First, it must be possible to order several different books in a single order and second, a customer must be able to distinguish between several orders placed the same day. To accomodate these requirements, they introduce a new attribute into the Orders table called *ordernum*, which uniquely identifies an order and therefore the customer placing the order. However, since several books could be purchased in a single order, *ordernum* and *isbn* are both needed to determine *qty* and *ship_date* in the Orders table.

Orders are assigned order numbers sequentially and orders that are placed later have higher order numbers. If several orders are placed by the same customer

on a single day, these orders have different order numbers and can thus be distinguished. The SQL DDL statement to create the modified Orders table follows:

```
CREATE TABLE Orders ( ordernum     INTEGER,
                      isbn         CHAR(10),
                      cid          INTEGER,
                      cardnum      CHAR(16),
                      qty          INTEGER,
                      order_date   DATE,
                      ship_date    DATE,
                      PRIMARY KEY (ordernum, isbn),
                      FOREIGN KEY (isbn) REFERENCES Books
                      FOREIGN KEY (cid) REFERENCES Customers )
```

The owner of B&N is quite happy with this design for Orders, but has realized something else. (DBDudes is not surprised; customers almost always come up with several new requirements as the design progresses.) While he wants all his employees to be able to look at the details of an order, so that they can respond to customer enquiries, he wants customers' credit card information to be secure. To address this concern, DBDudes creates the following view:

```
CREATE VIEW OrderInfo (isbn, cid, qty, order_date, ship_date)
       AS SELECT O.cid, O.qty, O.order_date, O.ship_date
          FROM   Orders O
```

The plan is to allow employees to see this table, but not Orders; the latter is restricted to B&N's Accounting division. We'll see how this is accomplished in Section 21.7.

3.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What is a relation? Differentiate between a relation schema and a relation instance. Define the terms *arity* and *degree* of a relation. What are domain constraints? **(Section 3.1)**
- What SQL construct enables the definition of a relation? What constructs allow modification of relation instances? **(Section 3.1.1)**
- What are *integrity constraints*? Define the terms *primary key constraint* and *foreign key constraint*. How are these constraints expressed in SQL? What other kinds of constraints can we express in SQL? **(Section 3.2)**

- What does the DBMS do when constraints are violated? What is *referential integrity*? What options does SQL give application programmers for dealing with violations of referential integrity? **(Section 3.3)**
- When are integrity constraints enforced by a DBMS? How can an application programmer control the time that constraint violations are checked during transaction execution? **(Section 3.3.1)**
- What is a *relational database query*? **(Section 3.4)**
- How can we translate an ER diagram into SQL statements to create tables? How are entity sets mapped into relations? How are relationship sets mapped? How are constraints in the ER model, weak entity sets, class hierarchies, and aggregation handled? **(Section 3.5)**
- What is a *view*? How do views support logical data independence? How are views used for security? How are queries on views evaluated? Why does SQL restrict the class of views that can be updated? **(Section 3.6)**
- What are the SQL constructs to modify the structure of tables and destroy tables and views? Discuss what happens when we destroy a view. **(Section 3.7)**

EXERCISES

Exercise 3.1 Define the following terms: *relation schema, relational database schema, domain, relation instance, relation cardinality,* and *relation degree.*

Exercise 3.2 How many distinct tuples are in a relation instance with cardinality 22?

Exercise 3.3 Does the relational model, as seen by an SQL query writer, provide physical and logical data independence? Explain.

Exercise 3.4 What is the difference between a candidate key and the primary key for a given relation? What is a superkey?

Exercise 3.5 Consider the instance of the Students relation shown in Figure 3.1.

1. Give an example of an attribute (or set of attributes) that you can deduce is *not* a candidate key, based on this instance being legal.
2. Is there any example of an attribute (or set of attributes) that you can deduce *is* a candidate key, based on this instance being legal?

Exercise 3.6 What is a foreign key constraint? Why are such constraints important? What is referential integrity?

Exercise 3.7 Consider the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In defined in Section 1.5.2.

1. List all the foreign key constraints among these relations.
2. Give an example of a (plausible) constraint involving one or more of these relations that is not a primary key or foreign key constraint.

Exercise 3.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

Emp(*eid:* integer, *ename:* string, *age:* integer, *salary:* real)
Works(*eid:* integer, *did:* integer, *pct_time:* integer)
Dept(*did:* integer, *dname:* string, *budget:* real, *managerid:* integer)

1. Give an example of a foreign key constraint that involves the Dept relation. What are the options for enforcing this constraint when a user attempts to delete a Dept tuple?
2. Write the SQL statements required to create the preceding relations, including appropriate versions of all primary and foreign key integrity constraints.
3. Define the Dept relation in SQL so that every department is guaranteed to have a manager.
4. Write an SQL statement to add John Doe as an employee with $eid = 101$, $age = 32$ and $salary = 15,000$.
5. Write an SQL statement to give every employee a 10 percent raise.
6. Write an SQL statement to delete the Toy department. Given the referential integrity constraints you chose for this schema, explain what happens when this statement is executed.

Exercise 3.9 Consider the SQL query whose answer is shown in Figure 3.6.

1. Modify this query so that only the *login* column is included in the answer.
2. If the clause WHERE *S.gpa >= 2* is added to the original query, what is the set of tuples in the answer?

Exercise 3.10 Explain why the addition of NOT NULL constraints to the SQL definition of the Manages relation (in Section 3.5.3) would not enforce the constraint that each department must have a manager. What, if anything, is achieved by requiring that the *ssn* field of Manages be non-*null*?

Exercise 3.11 Suppose that we have a ternary relationship R between entity sets A, B, and C such that A has a key constraint and total participation and B has a key constraint; these are the only constraints. A has attributes *a1* and *a2*, with *a1* being the key; B and C are similar. R has no descriptive attributes. Write SQL statements that create tables corresponding to this information so as to capture as many of the constraints as possible. If you cannot capture some constraint, explain why.

Exercise 3.12 Consider the scenario from Exercise 2.2, where you designed an ER diagram for a university database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.13 Consider the university database from Exercise 2.3 and the ER diagram you designed. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.14 Consider the scenario from Exercise 2.4, where you designed an ER diagram for a company database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.15 Consider the Notown database from Exercise 2.5. You have decided to recommend that Notown use a relational database system to store company data. Show the SQL statements for creating relations corresponding to the entity sets and relationship sets in your design. Identify any constraints in the ER diagram that you are unable to capture in the SQL statements and briefly explain why you could not express them.

Exercise 3.16 Translate your ER diagram from Exercise 2.6 into a relational schema, and show the SQL statements needed to create the relations, using only key and null constraints. If your translation cannot capture any constraints in the ER diagram, explain why.

In Exercise 2.6, you also modified the ER diagram to include the constraint that tests on a plane must be conducted by a technician who is an expert on that model. Can you modify the SQL statements defining the relations obtained by mapping the ER diagram to check this constraint?

Exercise 3.17 Consider the ER diagram that you designed for the Prescriptions-R-X chain of pharmacies in Exercise 2.7. Define relations corresponding to the entity sets and relationship sets in your design using SQL.

Exercise 3.18 Write SQL statements to create the corresponding relations to the ER diagram you designed for Exercise 2.8. If your translation cannot capture any constraints in the ER diagram, explain why.

Exercise 3.19 Briefly answer the following questions based on this schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
Dept(*did:* `integer`, *budget:* `real`, *managerid:* `integer`)

1. Suppose you have a view SeniorEmp defined as follows:

```
CREATE VIEW SeniorEmp (sname, sage, salary)
       AS SELECT E.ename, E.age, E.salary
          FROM   Emp E
          WHERE  E.age > 50
```

   Explain what the system will do to process the following query:

```
SELECT S.sname
FROM   SeniorEmp S
WHERE  S.salary > 100,000
```

2. Give an example of a view on Emp that could be automatically updated by updating Emp.
3. Give an example of a view on Emp that would be impossible to update (automatically) and explain why your example presents the update problem that it does.

Exercise 3.20 Consider the following schema:

Suppliers(*sid:* integer, *sname:* string, *address:* string)
Parts(*pid:* integer, *pname:* string, *color:* string)
Catalog(*sid:* integer, *pid:* integer, *cost:* real)

The Catalog relation lists the prices charged for parts by Suppliers. Answer the following questions:

- Give an example of an updatable view involving one relation.
- Give an example of an updatable view involving two relations.
- Give an example of an insertable-into view that is updatable.
- Give an example of an insertable-into view that is not updatable.

PROJECT-BASED EXERCISES

Exercise 3.21 Create the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In in Minibase.

Exercise 3.22 Insert the tuples shown in Figures 3.1 and 3.4 into the relations Students and Enrolled. Create reasonable instances of the other relations.

Exercise 3.23 What integrity constraints are enforced by Minibase?

Exercise 3.24 Run the SQL queries presented in this chapter.

BIBLIOGRAPHIC NOTES

The relational model was proposed in a seminal paper by Codd [187]. Childs [176] and Kuhns [454] foreshadowed some of these developments. Gallaire and Minker's book [296] contains several papers on the use of logic in the context of relational databases. A system based on a variation of the relational model in which the entire database is regarded abstractly as a single relation, called the *universal relation*, is described in [746]. Extensions of the relational model to incorporate *null* values, which indicate an unknown or missing field value, are discussed by several authors; for example, [329, 396, 622, 754, 790].

Pioneering projects include System R [40, 150] at IBM San Jose Research Laboratory (now IBM Almaden Research Center), Ingres [717] at the University of California at Berkeley, PRTV [737] at the IBM UK Scientific Center in Peterlee, and QBE [801] at IBM T. J. Watson Research Center.

A rich theory underpins the field of relational databases. Texts devoted to theoretical aspects include those by Atzeni and DeAntonellis [45]; Maier [501]; and Abiteboul, Hull, and Vianu [3]. [415] is an excellent survey article.

Integrity constraints in relational databases have been discussed at length. [190] addresses semantic extensions to the relational model, and integrity, in particular referential integrity. [360] discusses semantic integrity constraints. [203] contains papers that address various aspects of integrity constraints, including in particular a detailed discussion of referential integrity. A vast literature deals with enforcing integrity constraints. [51] compares the cost

of enforcing integrity constraints via compile-time, run-time, and post-execution checks. [145] presents an SQL-based language for specifying integrity constraints and identifies conditions under which integrity rules specified in this language can be violated. [713] discusses the technique of integrity constraint checking by query modification. [180] discusses real-time integrity constraints. Other papers on checking integrity constraints in databases include [82, 122, 138, 517]. [681] considers the approach of verifying the correctness of programs that access the database instead of run-time checks. Note that this list of references is far from complete; in fact, it does not include any of the many papers on checking recursively specified integrity constraints. Some early papers in this widely studied area can be found in [296] and [295].

For references on SQL, see the bibliographic notes for Chapter 5. This book does not discuss specific products based on the relational model, but many fine books discuss each of the major commercial systems; for example, Chamberlin's book on DB2 [149], Date and McGoveran's book on Sybase [206], and Koch and Loney's book on Oracle [443].

Several papers consider the problem of translating updates specified on views into updates on the underlying table [59, 208, 422, 468, 778]. [292] is a good survey on this topic. See the bibliographic notes for Chapter 25 for references to work querying views and maintaining materialized views.

[731] discusses a design methodology based on developing an ER diagram and then translating to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [513, 514].

Database Management Systems, known for its practical emphasis and comprehensive coverage, has quickly become one of the leading texts for database courses. The third edition features new material on database application development, with a focus on Internet applications. The hands-on approach introduces students to current standards, including JDBC, XML, and 3-tier application architectures. A new, flexible organization allows instructors to teach either an applications-oriented course or an introductory systems-oriented course. The revised "part" organization with new overview chapters makes it easy to select the chapters you need; in-depth chapters within each part can be optional.

This very current new edition also features pedagogical improvements (e.g., chapter objectives, review questions) and updated and extended discussions of data mining, database tuning wizards, decision support, information retrieval, Internet security, object-oriented databases, transaction processing, and XML data management. The coverage has been revised and expanded throughout to reflect the new SQL:1999 standard.

The new website **(www.mhhe.com/ramakrishnan)** includes extensive instructor resources:

- Exercise solutions
- Lecture slides in PowerPoint, Postscript, and PDF formats
- Notes on setting up and using Oracle, DB2, and SQL Server
- Online solutions and sample databases for all SQL exercises
- Programming exercises and solutions for database application development
- New online instructor's guide with chapter summaries and teaching tips
- Online chapter on QBE; online end-to-end Case Study

Raghu Ramakrishnan is Professor of Computer Science at the University of Wisconsin, Madison, and a founder and CTO of QUIQ. He is a Fellow of the ACM. Johannes Gehrke is Assistant Professor of Computer Science at Cornell University.

"The third edition of *Database Management Systems* continues the excellent tradition of previous editions. The mixture of theory and practice, along with interesting examples, makes the book engaging and easy to read. SQL is explained effectively—important features are covered without overly complicating the presentation with SQL's arcane details. I am particularly impressed with the well-chosen review questions and exercises, and the extensive bibliography. An excellent addition to any database student's or practitioner's library!" *Jim Melton, Editor of ISO/IEC 9075 (SQL), Oracle Corporation*

ISBN-10: 0-07-246563-8

McGraw-Hill Higher Education
A Division of The McGraw-Hill Companies