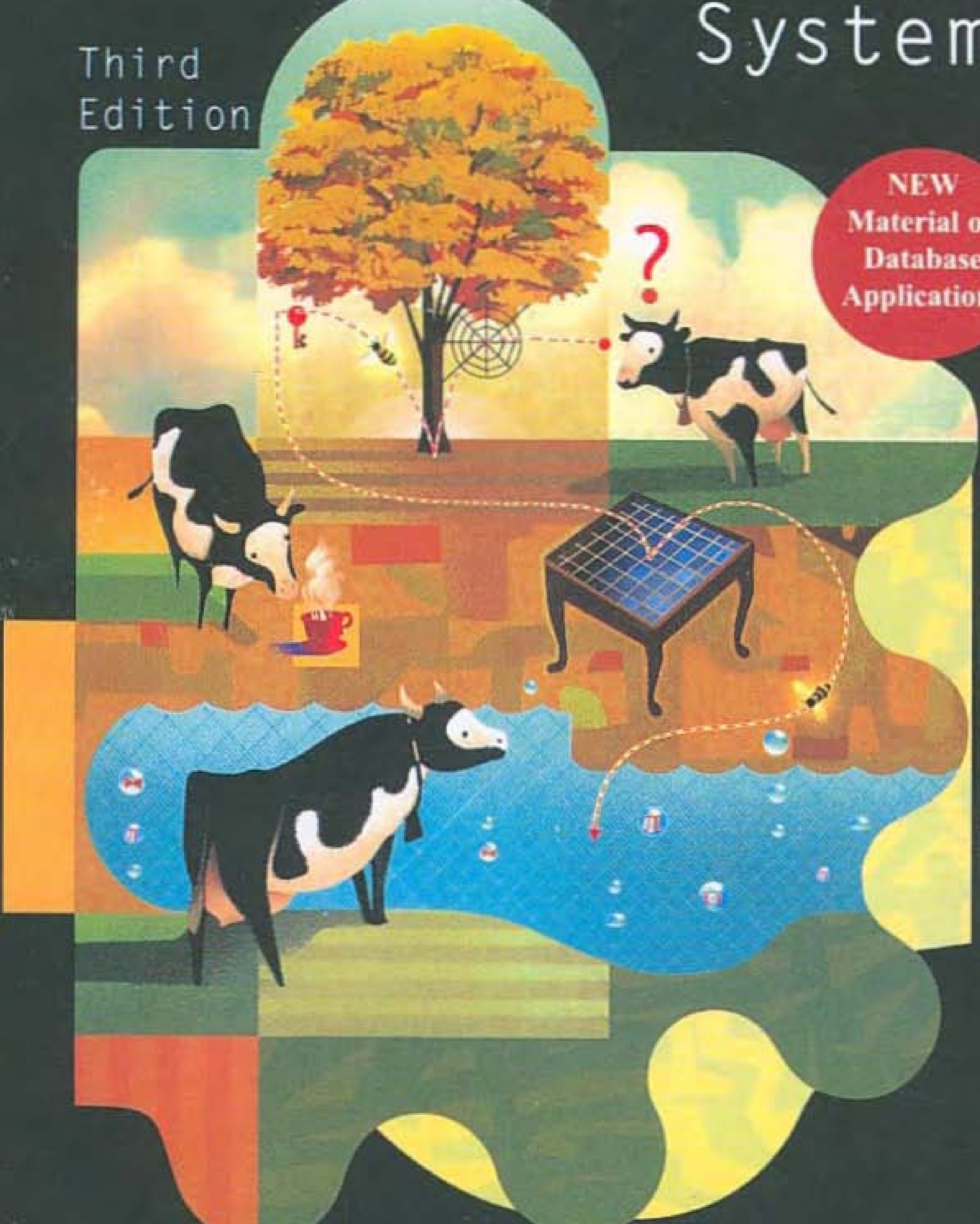
Database Management Systems

Third Edition



NEW Material on Database Applications

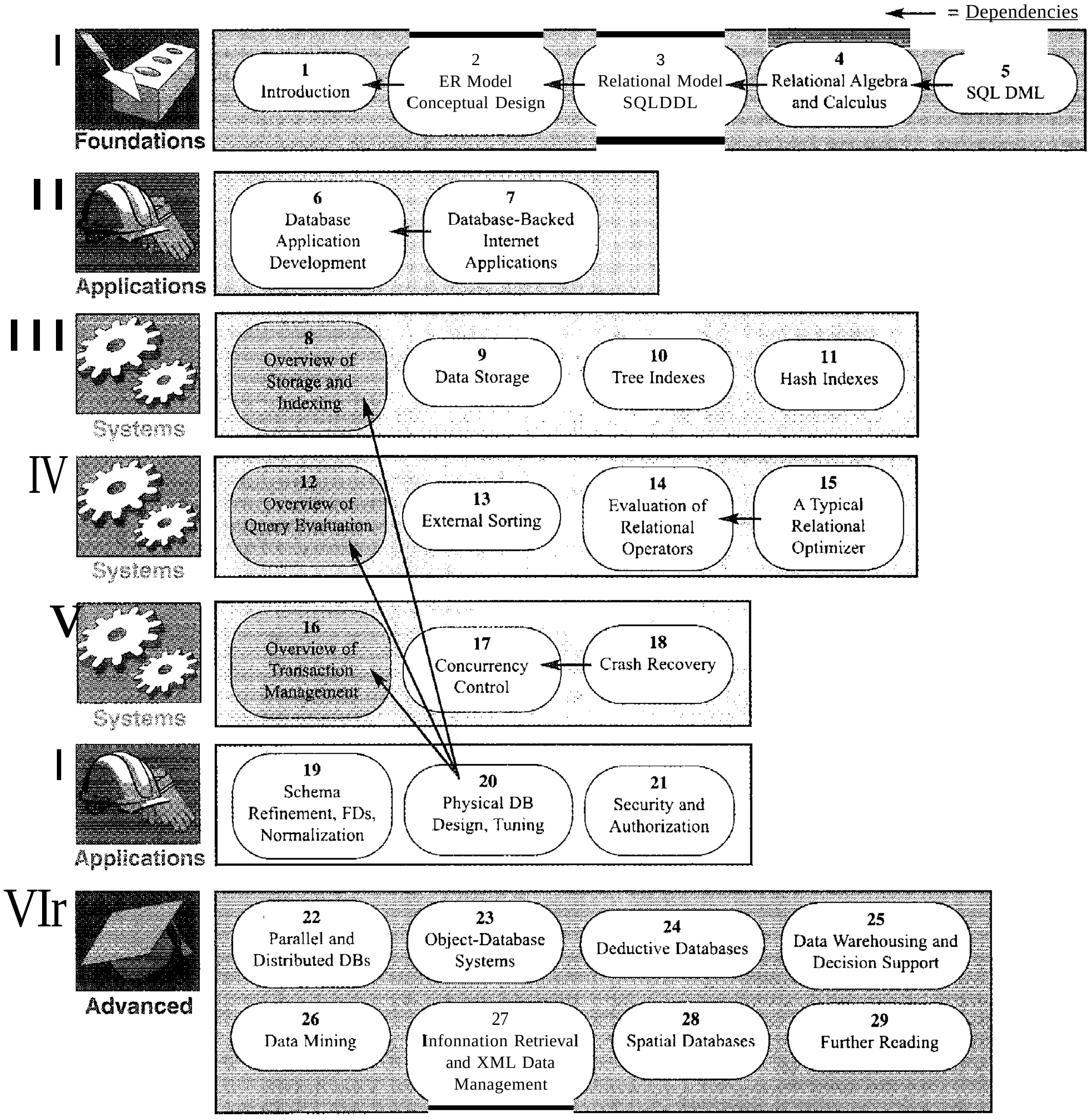
Ramakrishnan • Gehrke

It's your choice! New Modular Organization!

Applications emphasis: A course that covers the principles of database systems and emphasizes how they are used in developing data-intensive applications.

Systems emphasis: A course that has a strong systems emphasis and assumes that students have good programming skills in C and C++.

Hybrid course: Modular organization allows you to teach the course with the emphasis you want.



= Dependencies
I
Foundations
1 Introduction
2 ER Model Conceptual Design
3 Relational Model SQLDDL
4 Relational Algebra and Calculus
5 SQL DML
II
Applications
6 Database Application Development
7 Database-Backed Internet Applications
III
Systems
8 Overview of Storage and Indexing
9 Data Storage
10 Tree Indexes
11 Hash Indexes
IV
Systems
12 Overview of Query Evaluation
13 External Sorting
14 Evaluation of Relational Operators
15 A Typical Relational Optimizer
V
Systems
16 Overview of Transaction Management
17 Concurrency Control
18 Crash Recovery
I
Applications
19 Schema Refinement, FDs, Normalization
20 Physical DB Design, Tuning
21 Security and Authorization
VIr
Advanced
22 Parallel and Distributed DBs
23 Object-Database Systems
24 Deductive Databases
25 Data Warehousing and Decision Support
26 Data Mining
27 Infonnation Retrieval and XML Data Management
28 Spatial Databases
29 Further Reading

DATABASE MANAGEMENT SYSTEMS

DATABASE MANAGEMENT SYSTEMS

Third Edition

Raghu Ramakrishnan
University of Wisconsin
Madison, Wisconsin, USA

●

Johannes Gehrke
Cornell University
Ithaca, New York, USA

Mc Graw Hill

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogota 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
International Edition 2003

10 09 08 07 06 05 04 03
20 09 08 07 06 05 04
CTF BJE

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. 1. Gehrke, Johannes. II. Title.
QA76.9.D3 R237 2003
005.74—dc21 2002075205
CIP

When ordering this title, use ISBN 0-07-123151-X

Printed in Singapore

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 llsed 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 empha.sis. 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.

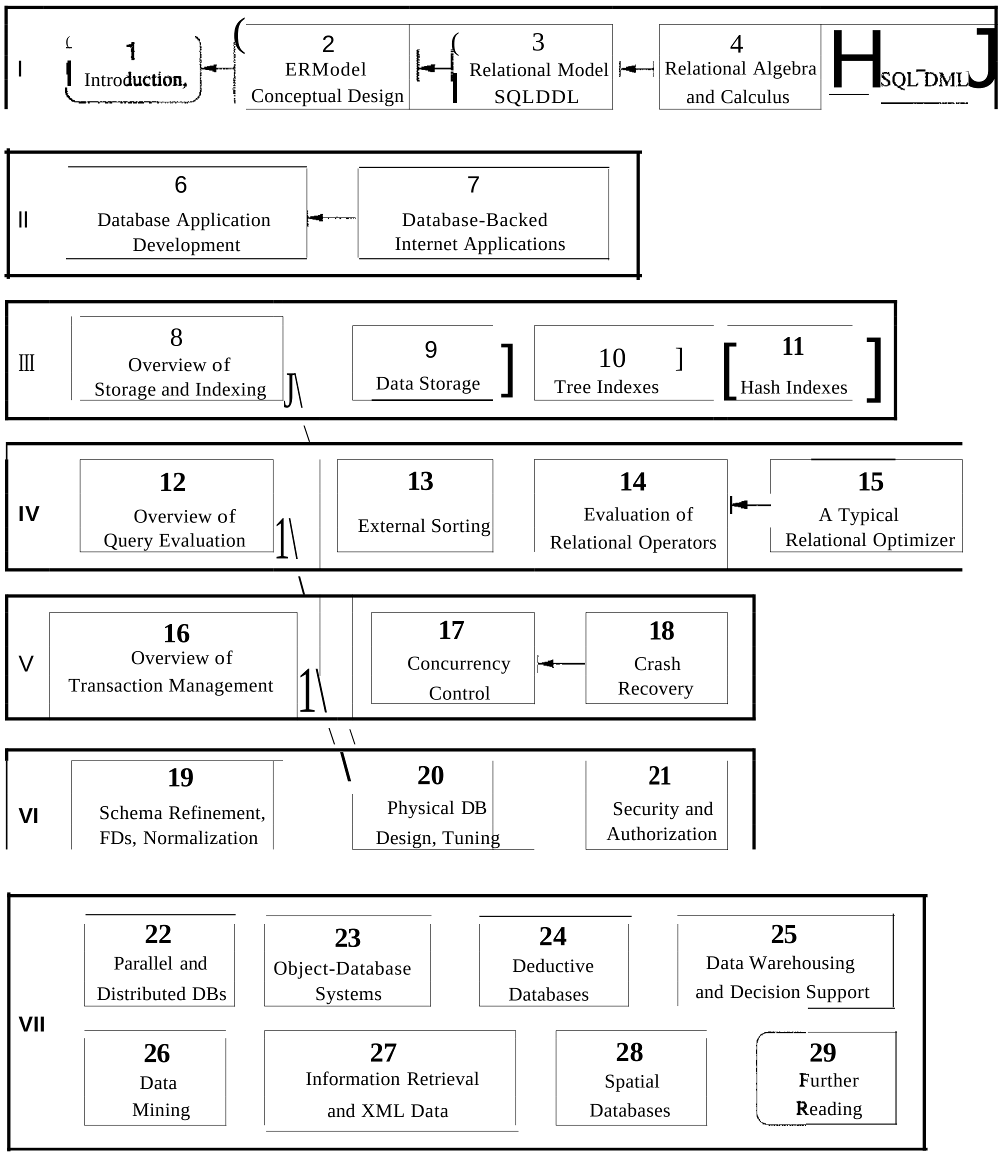

I
1 Introduction
2 ERModel Conceptual Design
3 Relational Model SQLDDL
4 Relational Algebra and Calculus
5 SQL DML
II
6 Database Application Development
7 Database-Backed Internet Applications
III
8 Overview of Storage and Indexing
9 Data Storage
10 Tree Indexes
11 Hash Indexes
IV
12 Overview of Query Evaluation
13 External Sorting
14 Evaluation of Relational Operators
15 A Typical Relational Optimizer
V
16 Overview of Transaction Management
17 Concurrency Control
18 Crash Recovery
VI
19 Schema Refinement, FDs, Normalization
20 Physical DB Design, Tuning
21 Security and Authorization
VII
22 Parallel and Distributed DBs
23 Object-Database Systems
24 Deductive Databases
25 Data Warehousing and Decision Support
26 Data Mining
27 Information Retrieval and XML Data
28 Spatial Databases
29 Further Reading

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 llsed 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 UvV-Madison. David De\Vitt 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. vVriting 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.

lowe 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 eomments 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 GrenadeI', 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 Vue 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 JVlelton 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 Sueiu were kind enough to let us adapt some of their lectures on XML. Mike Carey offered input on Web services.

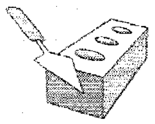
Emily Lupash at McGraw-Hill has been a source of constant support and encouragement. She coordinated extensive reviews from Ming Wang at Embry-Riddle Aeronautical University, Cheng Hsu at RPI, Paul Bergstein at Univ. of Massachusetts, Archana Sathaye at SJSU, Bharat Bhargava at Purdue, John Fendrich 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 QWEHTY 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 alIlount 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 systenls 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 progTamming 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 DBIvIS 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 corc chapters of this hook:

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? \Vhat factors must be considered in deciding how to organize the stored data? How can ,ve 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, la, 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 (rnany, 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

2Informix 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 rnanner. 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 ('1'B) is 1024 CBs, 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 impOl'tant 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 sig11ificant 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 DBlVIS 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 *nevCT* 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. \Ve 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*	*lZogin*	*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 completeo----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 call 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 coneeptual schemas. We discuss the DDL facilities of the Inost 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.

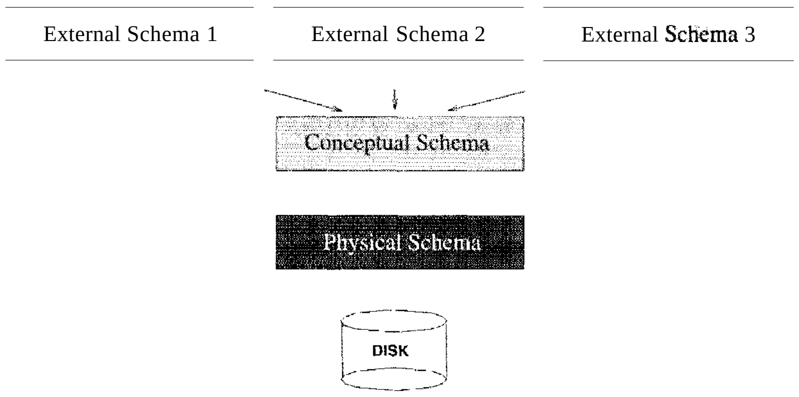


Figure 1.2 Levels of Abstraction in a DBMS

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 exalnple, 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(*rid:* 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 \vith 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 1234567
2. What is the average salary of professors who teach course CS5647
3. How many students are enrolled in CS5647
4. What fraction of students in CS564 received a grade better than B7
5. Is any student with a CPA less than 3.0 enrolled in CS5647

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 Dl\ilL, 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 *anyone 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 progTam 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 va,riety 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 DBIVIS 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 comma,nels 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

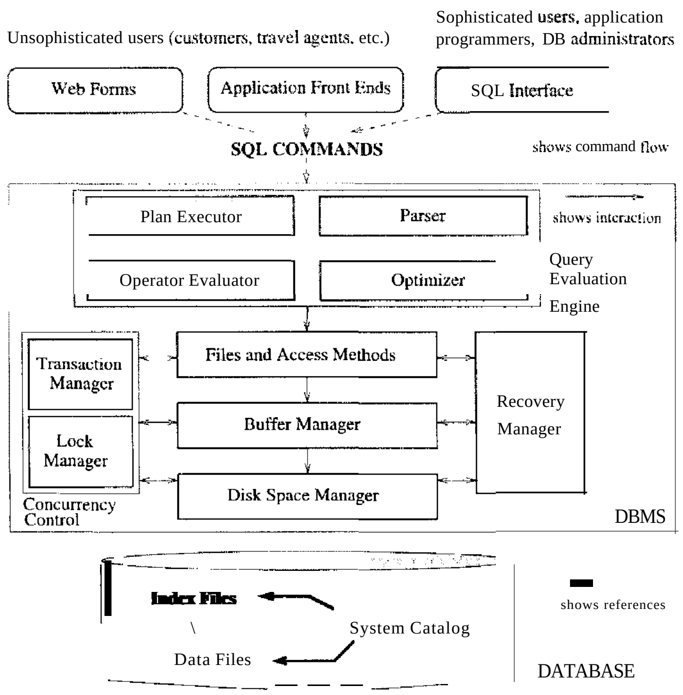


Figure 1.3 Architecture of a DBMS

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 cQIlsidered 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 comprornise 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 *DB.I\!IS* 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? \Vhat 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? \Vhat 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 aS 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?

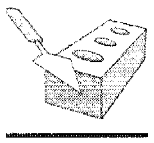
PROJECT-BASED EXERCISES

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 databa,se 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 (levelop an initial database design. It provides useful eoncepts that allow us to move fronl an informal description of what users want 1'rom 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.

\Ve begin with an overview of database design in Section 2.1 in order to motivate our discussion of the ER model. \Vithin 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 ill 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 DBlVIS 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 frQm other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the rnanagel' 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 *mting,* 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.

iTo 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.

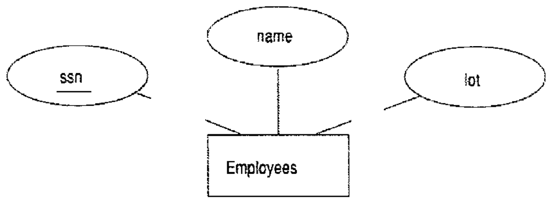


Figure 2.1 The Employees Entity Set

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.

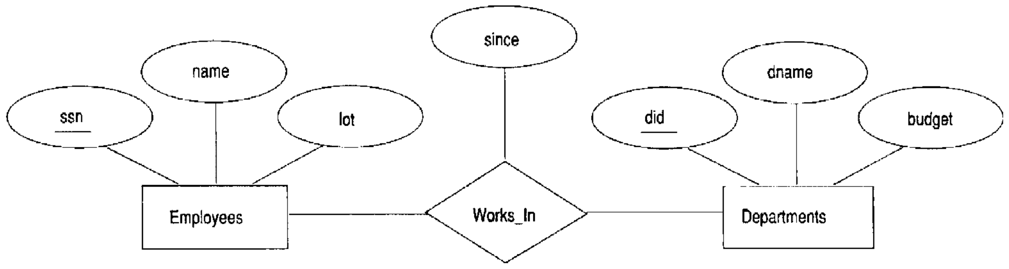


Figure 2.2 The Works_In Relationship Set

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 *d'id.* 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-te-many' and 'total participation' comments in the figure are discussed later, when we discuss integrity constraints.)

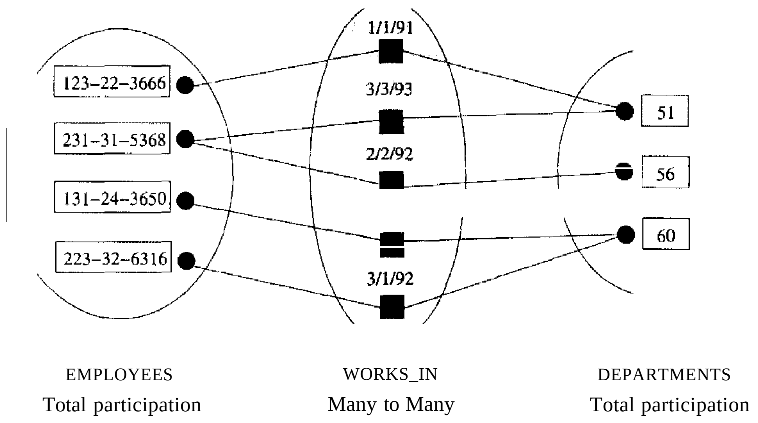


Figure 2.3 An Instance of the Works_In Relationship Set

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.ln2, is shown in Figure 2.4.

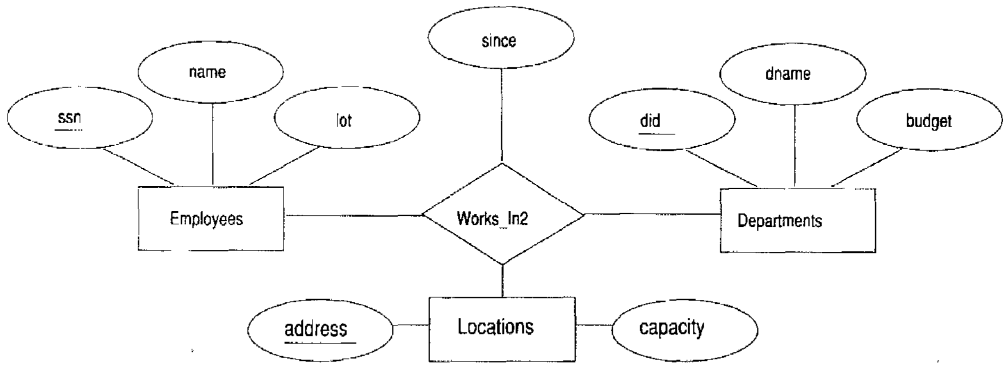


Figure 2.4 A Ternary Relationship Set

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 shown in Figure 2.5. Since employees report to other employees, every relationship in Reports_To is of the form $(emp_1, emp2)$, where both *empl* and *empz* are entities in Employees. However, they play different roles: *ernpl* reports to the managing employee *emp2,* 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.*

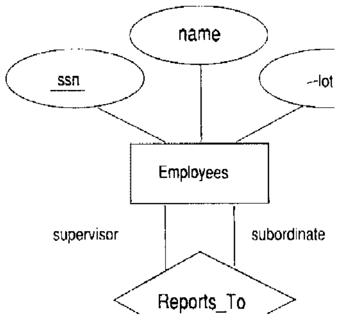


Figure 2.5 The Reports_To Relationship Set

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 lise.

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 vVorks_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.

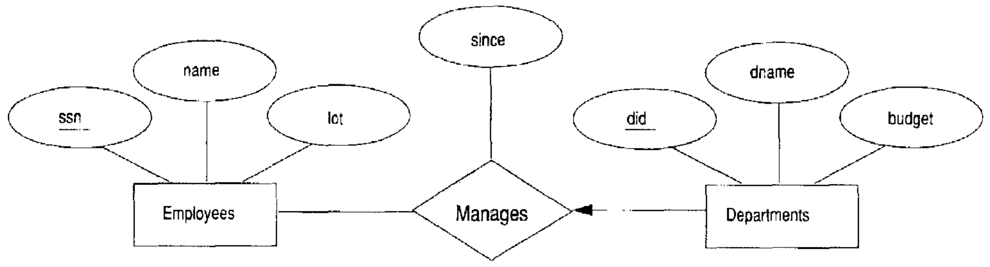


Figure 2.6 Key Constraint on Manages

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.

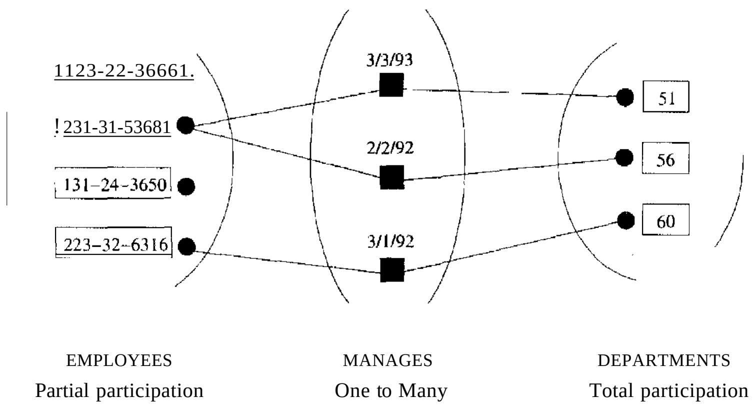


Figure 2.7 An Instance of the Manages Relationship Set

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 lVlanages 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.

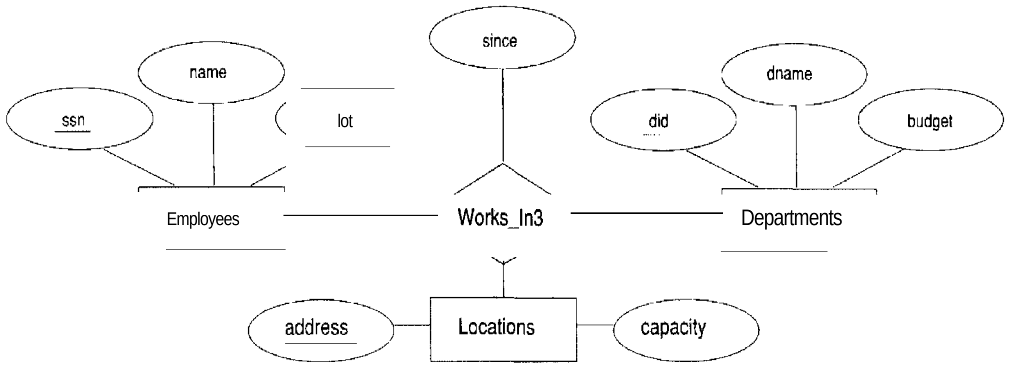


Figure 2.8 A Ternary Relationship Set with Key Constraints

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 Inanagel'. 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

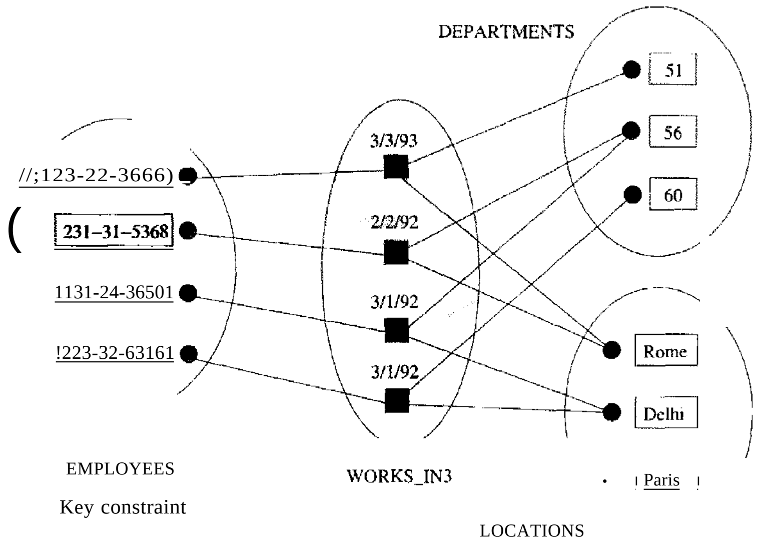


Figure 2.9 An Instance of Works_In3

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

Revisiting the Works..ln 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..ln is total. The ER diagram in Figure 2.10 shows both the Manages and Works..ln 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.

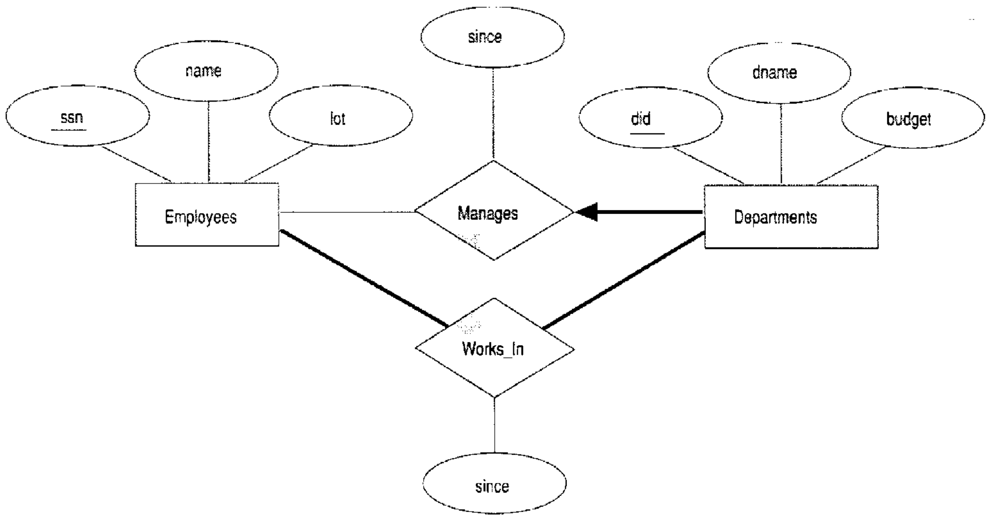


Figure 2.10 Manages and Works_In

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 2.1.1. 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.

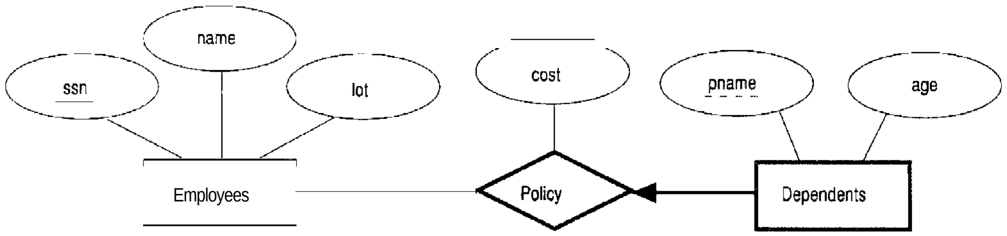


Figure 2.11 A Weak Entity Set

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 ContracLEmps 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 ContracLEmps.

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 ContracLEmps 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 SenioLEmps. We can rnodify Figure 2.12 to reflect this change by adding a second ISA node as a child of Employees and making SenioLEmps a child of this node. Each of these entity sets might be classified further, creating a multilevel ISA hierarchy.

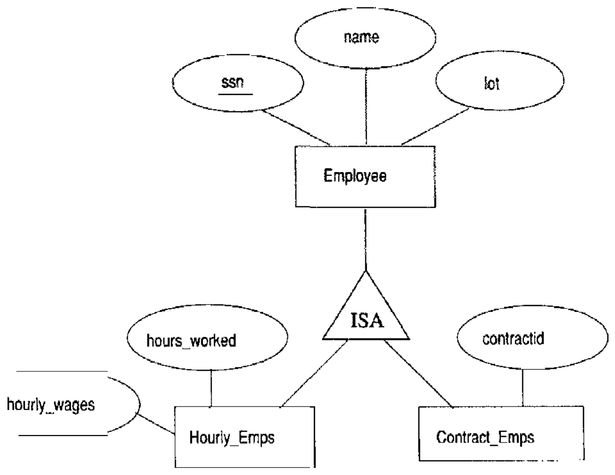


Figure 2.12 Class Hierarchy

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 ContracLEmps 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 ContracLEmps entity? Intuitively, no. Can he be both a ContracLEmps 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 ContracLEmps 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.

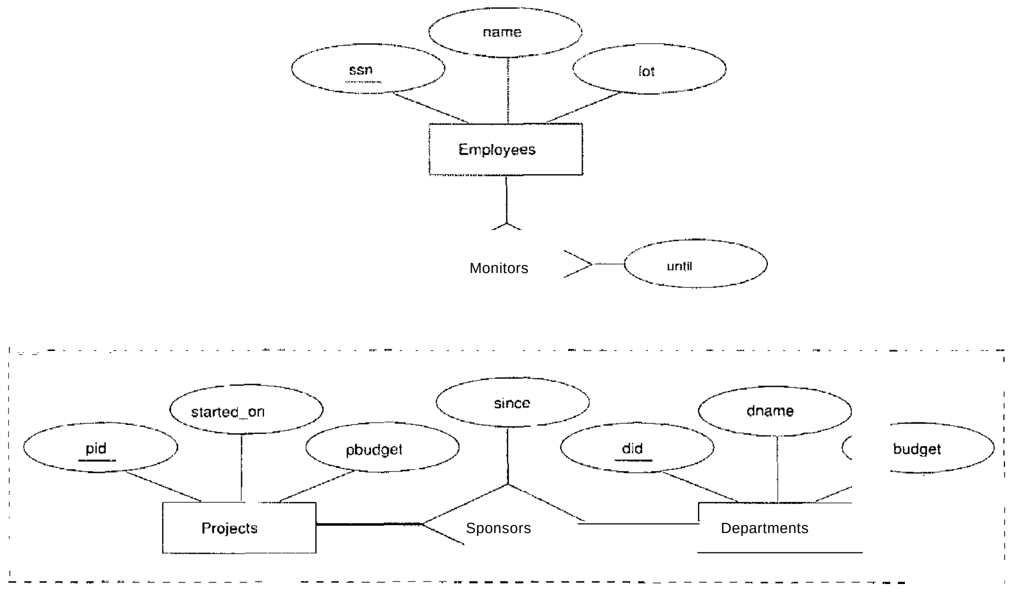


Figure 2.13 Aggregation

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 arc the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

\Ve 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 WorksJ:n4) shown in Figure 2.14.

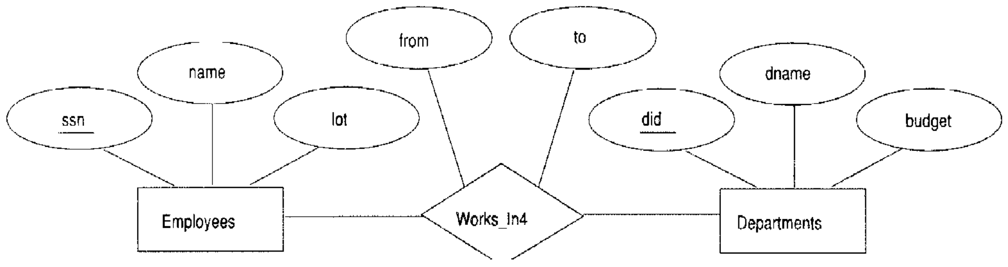


Figure 2.14 The Works_In4 Relationship Set

It differs from the \Vorks_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.

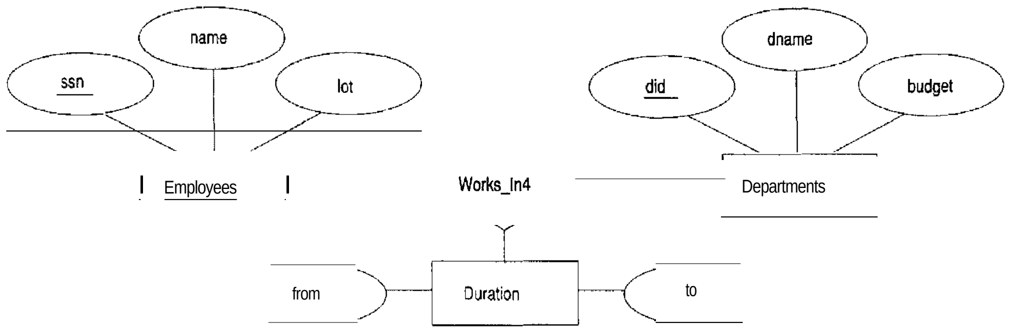


Figure 2.15 The Works_In4 Relationship Set

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.

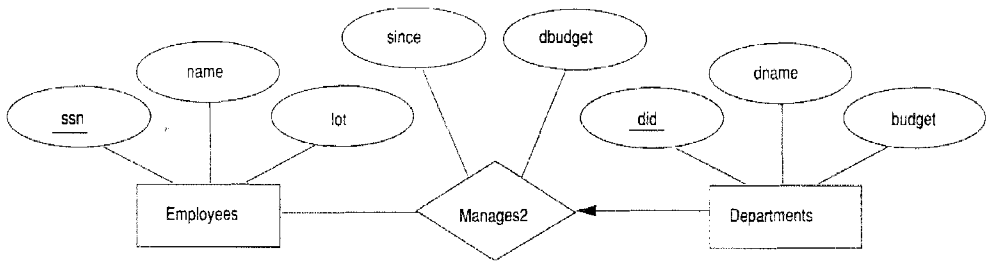


Figure 2.16 Entity versus Relationship

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

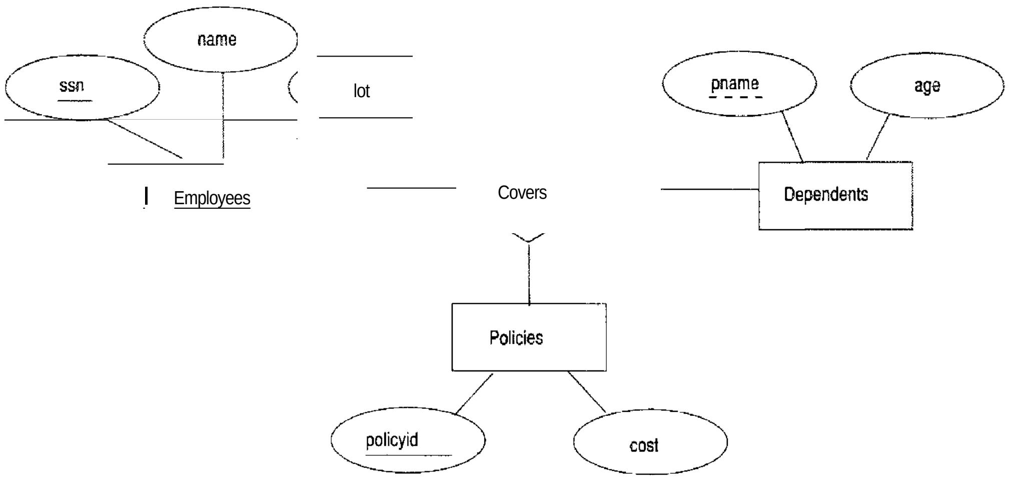


Figure 2.17 Policies as an Entity Set

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.

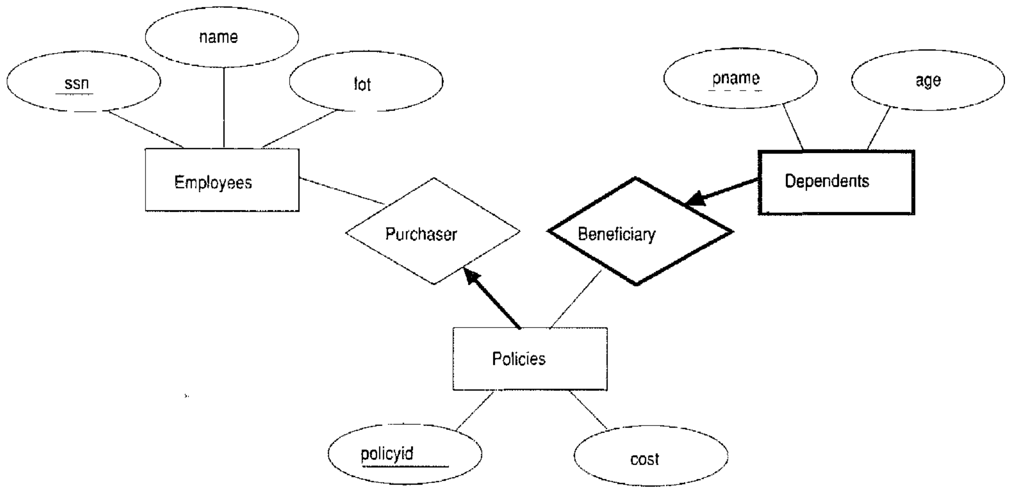


Figure 2.18 Policy Revisited

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 ternal'Y 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.

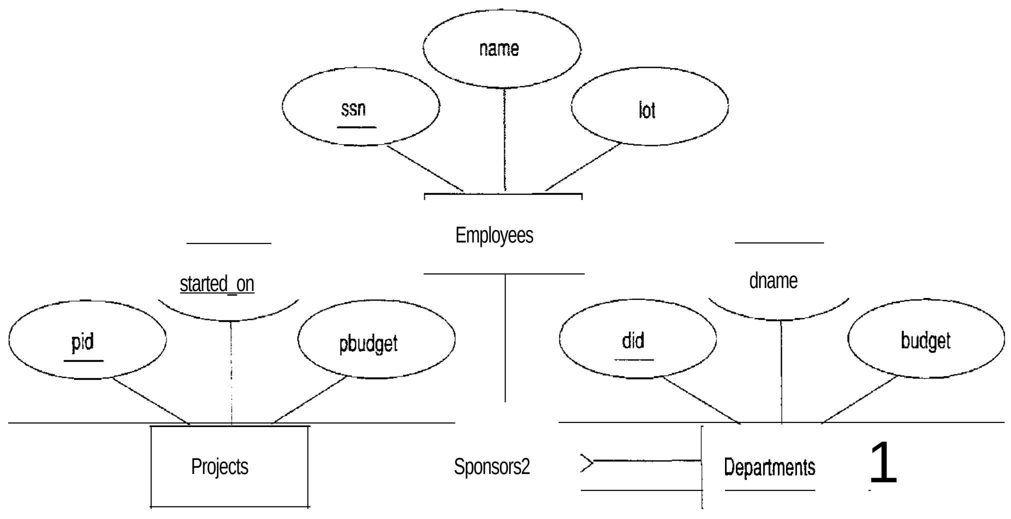


Figure 2.19 Using a Ternary Relationship instead of Aggregation

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 multipIe 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 systern 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, fOTe'ign keys, null-allowed fields)* and that reflect physical design choices (e.g., indexed fields) can be modeled ill UML database diagrams.

UML's **component** diagrams describe storage aspects of the database, such as *tablespaces* and *database pa,titions)*, 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.

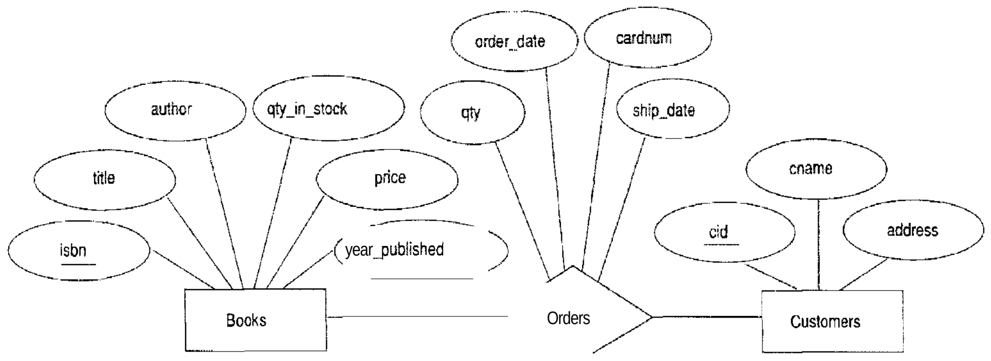


Figure 2.20 ER Diagram of the Initial Design

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 ha,ndlecl by crea.ting 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: \Vhether 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 constmint. 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.

BffiLIOGRAPHIC NOTES

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

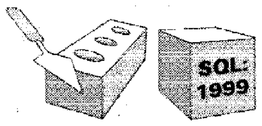
The ER model was proposed by Chen [172], and extensions have been proposed in a number of subsequent papers. Generalization and aggregation were introduced in [693]. [390, 589] contain good surveys of semantic data models. Dynamic and temporal aspects of semantic data models are discussed in [749].

[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.

-----vVebster'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 SQLfeatures 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 DC-Berkeley by the mid-197Gs, 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* (lGs), enable the DBIviS 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 les. 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.

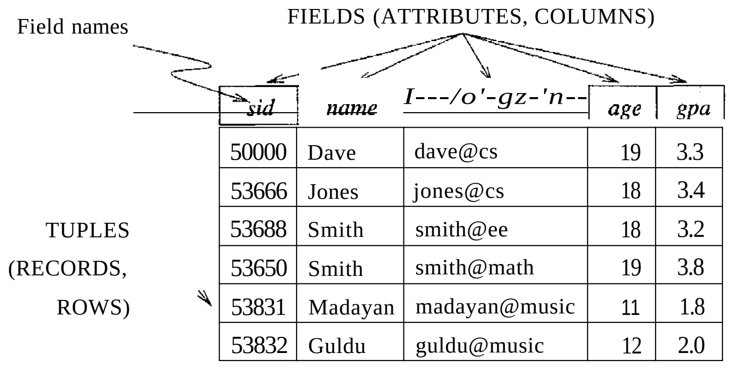
\Ve use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

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 81



sid	*name*	*I---/o'-gz-'n--*	*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 81 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

s'id	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	gllldll@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 81 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, ..., *In:Dn)* be a relation schema, and for each f_i, $1 \leq$ i $\leq n$, let *Dami* 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:

$$\{ \langle f_1 : d_l , \quad ,In: d_n) \mid d_l \in Daml' \ \ldots \ ,d_n \in Damn\}$$

The angular brackets ( ) identify the fields of a tuple. Using this notation, the first Students tuple shown in Figure 3.1 is written as *(sid:* 50000, *name:* Dave, *login:* dave@cs, *age:* 19, *gpa:* 3.3). The curly brackets {...} denote a set (of tuples, in this definition). The vertical bar I should be read 'such that,' the symbol E 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 discllssed 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 81 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 81 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 (Ie)** 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 Ies, 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. \Vhen 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 les 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

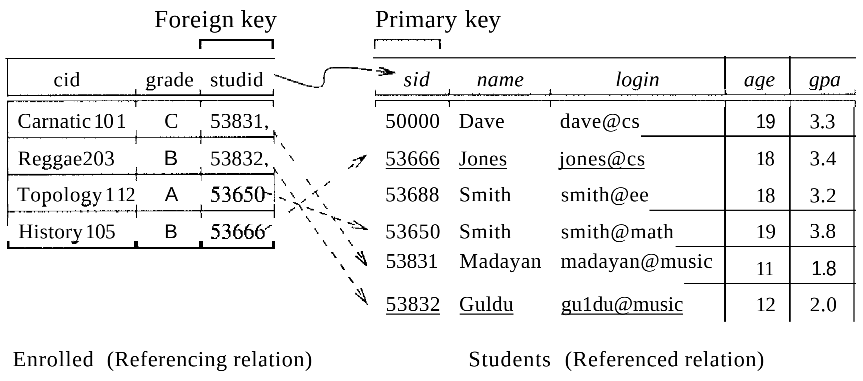
Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. 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, *gTade:* 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 cornpatible 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.



cid	grade	studid
Carnatic101	C	53831
Reggae203	B	53832
Topology112	A	53650
History105	B	53666

Enrolled (Referencing relation)

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	gu1du@music	12	2.0

Students (Referenced relation)

Figure 3.4 Referential Integrity

If we try to insert the tuple (55555, *Artl04, A)* into *E1,* the Ie is violated because there is no tuple in 51 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 51, we violate the foreign key constraint because the tuple (53666, *Historyl05, B)* in *El* 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 partnerT' This situation is handled in SQL by using a special value called null. The use of *null* in a field of a tuple rneans 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 *st'udid* 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 82 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 Ie 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 51 of Students shown in Figure 3.1. The following insertion violates the primary key constraint because there is already a tuple with the *s'id* 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 clornain, primary key or unique constraints. However, an update can cause violations, sirnilar 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 81 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 *stud'id* 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 val'ue 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, dd),
                        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,
                        honorsCHAR(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 (dd)
                        FOREIGN KEY (grader) REFERENCES Students (sid))
```

vVhenever 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 51

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."id* 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 81 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 Sand 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, *(Smith, Topology112).*

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

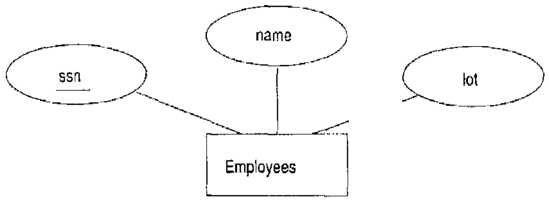


Figure 3.8 The Employees Entity Set

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.

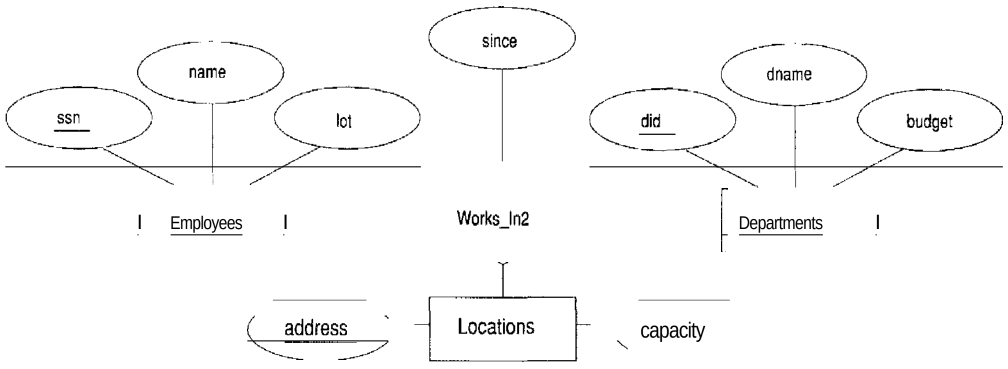


Figure 3.10 A Ternary Relationship Set

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 (8sn, 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 *n'ull* values. Because these fields are part of the primary key for \Vorks_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 WorksJn. 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

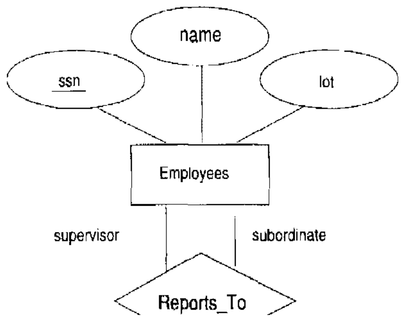


Figure 3.11 The Reports_To Relationship Set

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 somem of them are linked via arrows in the ER diagTam, the key for anyone 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-

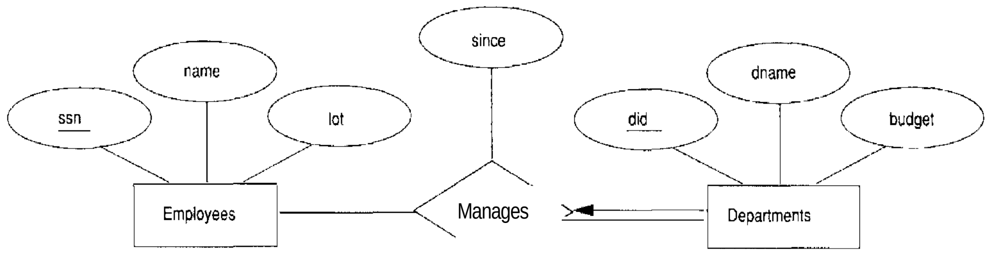


Figure 3.12 Key Constraint on Manages

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 adyantage 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 Inanager 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 DepLMgr relation that captures the information in both Departments and Manages, illustrates the second approach to translating relationship sets with key constraints:

```
CREATE TABLE DepLMgr ( 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 anyone 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:

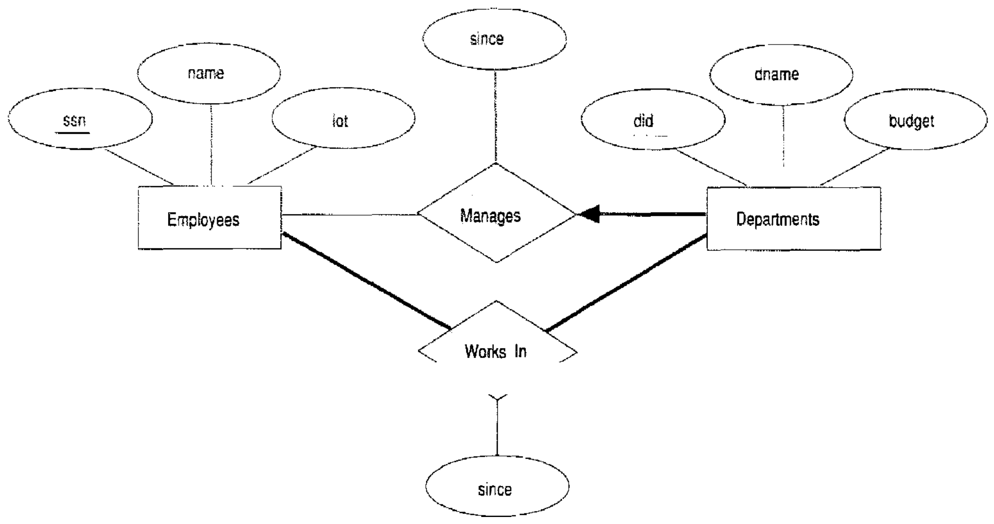


Figure 3.13 Manages and WorksJn

```
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.

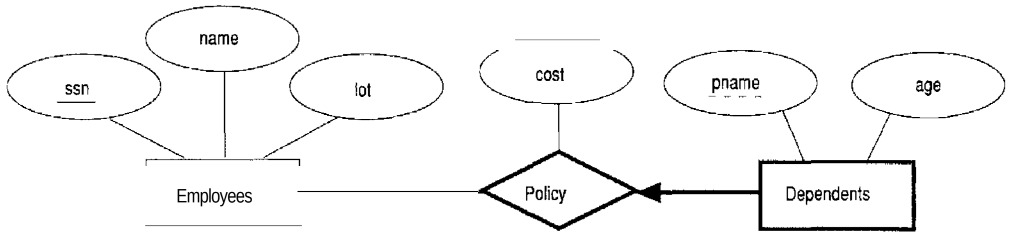


Figure 3.14 The Dependents Weak Entity Set

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:

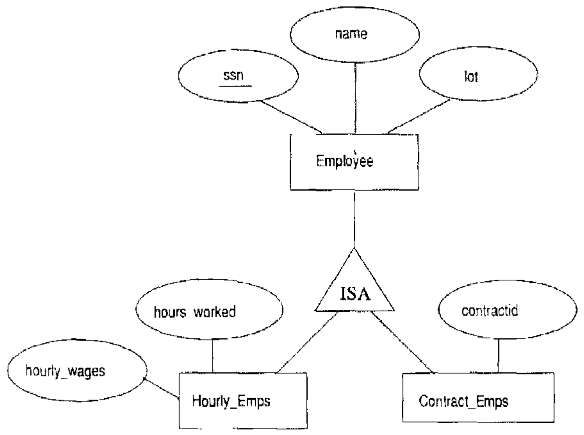


Figure 3.15 Class Hierarchy

1. We can map each of the entity sets Employees, Hourly_Emps, and ContracLEmps to a distinct relation. The Employees relation is created as in Section 2.2. We discuss Hourly_Emps here; ContracLEmps 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 ContracLEmps. 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 ContracLEmps, 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 ContracLEmps 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,

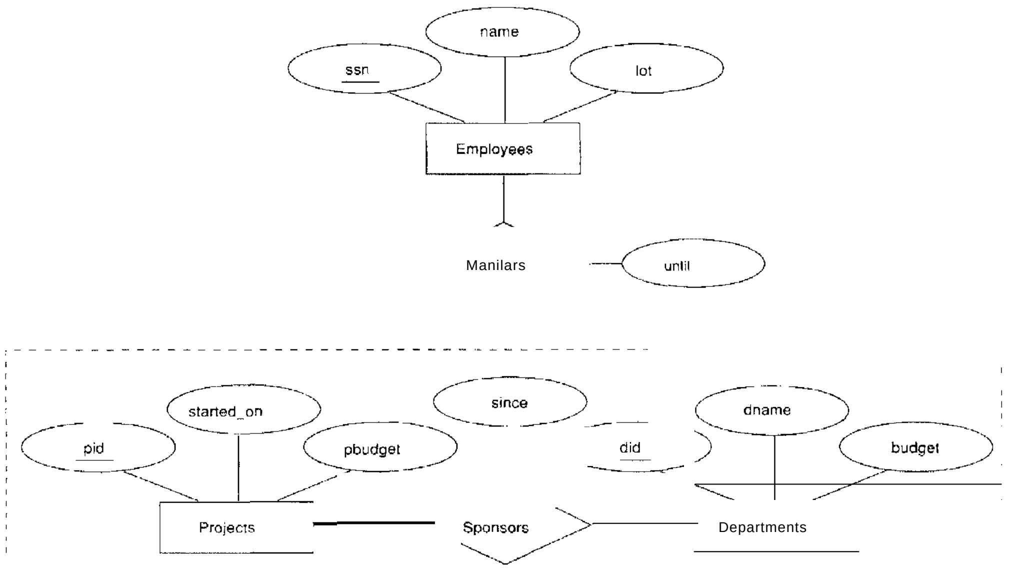


Figure 3.16 Aggregation

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 *(88n)*, 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. Consicler the Sponsors relation. It has attributes *pid, did,* and *since;* and in general we need it (in addition to l\rlonitors) for two reasons:

1. \Ve have to record the descriptive attributes (in our example, *since)* of the Sponsors relationship.
2. Not every sponsorship has a monitor, and thus some $\langle pid, did\rangle$ 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 $\langle pid, did\rangle$ 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

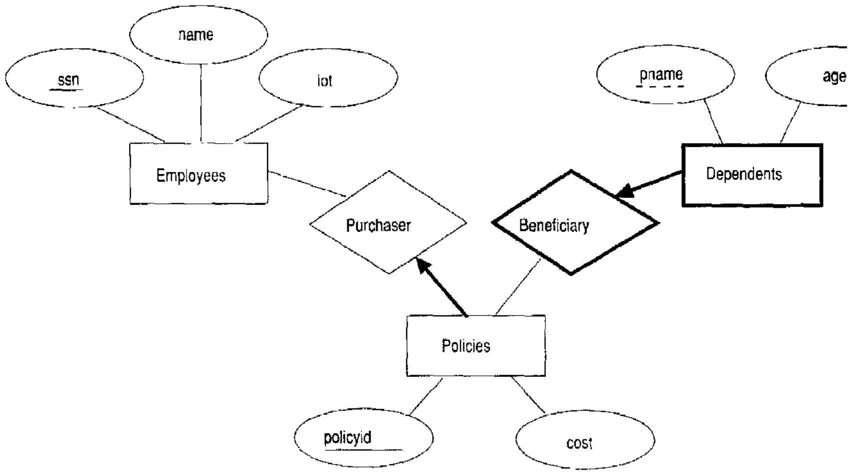


Figure 3.17 Policy Revisited

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 *(policyid, ssn)*, 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 *exte'mal* 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 GooclStudents.

3There 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 Chapt.er 5.

Updatable Views in SQL:1999 The Hew 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 OIle table in the FROM clause may be updatable under the new definition. Intuitively, we can update afield 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 (51234, 2.8) 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

vVhile 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)

	jyear	
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 53 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 *(8m:ith, 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 *d'isallow* 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,
                     yeaLpublished   INTEGER,
                     PRIMARY KEY (isbn))

CREATE TABLE Orders ( isbn         CHAR(10),
                      ciel         INTEGER,
                      carelnum     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 *c'id*. Because of this, a customer cannot order the same book OII 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 restrietion:

```
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 do·cument. 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),
                      dd            INTEGER,
                      cardnum       CHAR(16),
                      qty           INTEGER,
                      ordeLdate     DATE,
                      ship_date     DATE,
                      PRIMARY KEY (ordernum, isbn),
                      FOREIGN KEY (isbn) REFERENCES Books
                      FOREIGN KEY (dd) 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.ordeLdate, O.ship_date
           FROM   Orders 0
```

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)

- \Vhat does the DBMS do when constraints are violated? What is *referential 'integr-ity?* \Vhat 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 destray 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(<u>*eid:* integer</u>, *ename:* string, *age:* integer, *salary:* real)
> Works(<u>*eid:* integer</u>, <u>*did:* integer</u>, *pct_time:* integer)
> <u>Dept(did: integer</u>, *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, Kage, E.salary
          FROM   Emp E
          WHERE  Kage > 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 \vith 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 fTom 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].

4 RELATIONAL ALGEBRA AND CALCULUS

- ☛ What is the foundation for relational query languages like SQL? What is the difference between procedural and declarative languages?
- ☛ What is relational algebra, and why is it important?
- ☛ What are the basic algebra operators, and how are they combined to write complex queries?
- ☛ What is relational calculus, and why is it important?
- ☛ What subset of mathematical logic is used in relational calculus, and how is it used to write queries?
- ➽ Key concepts: relational algebra, select, project, union, intersection, cross-product, join, division; tuple relational calculus, domain relational calculus, formulas, universal and existential quantifiers, bound and free variables

Stand finn in your refusal to remain conscious during algebra. In real life, I assure you, there is no such thing as algebra.

—Fran Lebowitz, *Social Studies*

This chapter presents two formal query languages associated with the relational model. Query 'languages are specialized languages for asking questions, or queries, that involve the data in a database. After covering some preliminaries in Section 4.1, we discuss *relational algebra* in Section 4.2. Queries in relational algebra are composed using a collection of operators, and each query describes a step-by-step procedure for computing the desired answer; that is, queries are specified in an *operational* manner. In Section 4.3, we discuss *relational calculus*, in which a query describes the desired ans\ver without specifying how the answer is to be computed; this nonprocedural style of querying is called *declar-at'i'Ve*. We usually refer to relational algebra and relational calculus as algebra and calculus, respectively. We compare the expressive power of algebra and calculus in Section 4.4. These formal query languages have greatly influenced commercial query languages such as SQL, which we discuss in later chapters.

4.1 PRELIMINARIES

We begin by clarifying some important points about relational queries. The inputs and outputs of a query are relations. A query is evaluated using *instances* of each input relation and it produces an instance of the output relation. In Section 3.4, we used field names to refer to fields because this notation makes queries more readable. An alternative is to always list the fields of a given relation in the same order and refer to fields by position rather than by field name.

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances; and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances. This can be tedious and is really a secondary issue, because we can refer to fields by position anyway. On the other hand, field names make queries more readable.

Due to these considerations, we use the positional notation to formally define relational algebra and calculus. We also introduce simple conventions that allow intermediate relations to 'inherit' field names, for convenience.

We present a number of sample queries using the following schema:

> Sailors(sid: integer, *sname:* string, *rating:* integer, *age:* real)
> Boats(*bid:* integer, *bnarne:* string, *coloT:* string)
> Reserves(*sid:* integer, *bid:* integer, *day:* date)

The key fields are underlined, and the domain of each field is listed after the field name. Thus, *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations are referred to by name, or positionally, using the order in which they were just listed.

In several examples illustrating the relational algebra operators, we use the instances 81 and 82 (of Sailors) and *R1* (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively.

sid			
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.1 Instance *Sl* of Sailors

sid		rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Figure 4.2 Instance *S2* of Sailors

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 4.3 Instance *Rl* of Reserves

4.2 RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to *compose* operators to form a complex query-a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

4.2.1 Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation. Consider the instance of the Sailors relation shown in Figure 4.2, denoted as *52*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression

$$\sigma_{rating>8}(52)$$

evaluates to the relation shown in Figure 4.4. The subscript *rating*> 8 specifies the selection criterion to be applied while retrieving tuples.

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname,rating}(S2)$

The selection operator σ specifies the tuples to retain through a *selection condition.* In general, the selection condition is a Boolean combination (i.e., an expression using the logical connectives /\ and V) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <, <=, =, $\neq$, >=, or >. The reference to an attribute can be by position (of the form .i or i) or by name (of the form *.name* or *name).* The schema of the result of a selection is the schema of the input relation instance.

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using 1f. The expression

$$\pi_{sname,rafing(52)}$$

evaluates to the relation shown in Figure 4.5. The subscript *8na:me)rating* specifies the fields to be retained; the other fields are 'projected out.' The schema of the result of a projection is determined by the fields that are projected in the obvious way.

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi_{age}(S2)$$

evaluates to the relation shown in Figure 4.6. The irnportant point to note is that, although three sailors are aged 35, a single tuple with *age=35.0* appears in the result of the projection. This follm\'8 from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples,* leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

Since the result of a relational algebra expression is always a relation, we can substitute an expression wherever a relation is expected. For example, we can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(82))$$

produces the result shown in Figure 4.7. It is obtained by applying the selection to 82 (to get the relation shown in Figure 4.4) and then applying the projection.

age
35.0
55.5

Figure 4.6 $\pi_{age}(S2)$

sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

4.2.2 Set Operations

The following standard operations on sets are also available in relational algebra: *un'ion* (U), *intersection* (n), *set-difference* (—), and *cross-product* (x).

- **Union:** R U 8 returns a relation instance containing aU tuples that occur in *either* relation instance *R* or relation instance 8 (or both). Rand 8 must be *union-compatible,* and the schema of the result is defined to be identical to the schema of *R.*

  Two relation instances are said to be **union-compatible** if the following conditions hold:
  - they have the same number of the fields, and
  - corresponding fields, taken in order from left to right, have the same *domains.*

  Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of R U S inherit names from R, if the fields of R have names. (This assumption is implicit in defining the schema of R U S to be identical to the schema of R, as stated earlier.)

- **Intersection:** R n S returns a relation instance containing all tuples that occur in *both* Rand *S.* The relations Rand *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R.*

- Set-difference: $R-8$ returns a relation instance containing all tuples that occur in R but not in 8. The relations Rand 8 must be union-compatible, and the schema of the result is defined to be identical to the schema of R.
- Cross-product: R x 8 returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of 8 (in the same order as they appear in 8). The result of R x 8 contains Olle tuple *(1',* s) (the concatenation of tuples rand s) for each pair of tuples $r' \in R$, $s \in 8$. The cross-product opertion is sometimes called Cartesian product.

  We use the convention that the fields of R x 8 inherit names from the corresponding fields of Rand 8. It is possible for both Rand 8 to contain one or more fields having the same name; this situation creates a *naming confi'ict.* The corresponding fields in R x 8 are unnamed and are referred to solely by position.

In the preceding definitions, note that each operator can be applied to relation instances that are computed using a relational algebra (sub)expression.

We now illustrate these definitions through several examples. The union of 81 and 82 is shown in Figure 4.8. Fields are listed in order; field names are also inherited from 81. 82 has the same field names, of course, since it is also an instance of Sailors. In general, fields of 82 may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. TUples that appear in both 81 and 82 appear only once in 81 U82. Also, 81 uRI is not a valid operation because the two relations are not union-compatible. The intersection of 81 and 82 is shown in Figure 4.9, and the set-difference 81- 82 is shown in Figure 4.10.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0

Figure 4.8 31 U 52

The result of the cross-product 81 x Rl is shown in Figure 4.11. Because Rl and 81 both have a field named *sid,* by our convention on field names, the corresponding two fields in 81 x Rl are unnamed, and referred to solely by the position in which they appear in Figure 4.11. The fields in 81 x Rl have the same domains as the corresponding fields in Rl and 5'1. In Figure 4.11, *sid* is

sid	*sname*		
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.9 81 n 82

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 81 - 82

listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(sid)	*sname*	*rating*	*age*	*(sid)*	*bid*	*day*
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 81 x *R1*

4.2.3 Renaming

We have been careful to adopt field name conventions that ensure that the result of a relational algebra expression inherits field names from its argument (input) relation instances in a natural way whenever possible. However, name conflicts can arise in some cases; for example, in 81 x Rl. It is therefore convenient to be able to give names explicitly to the fields of a relation instance that is defined by a relational algebra expression. In fact, it is often convenient to give the instance itself a name so that we can break a large algebra expression into smaller pieces by giving names to the results of subexpressions.

We introduce a renaming operator *p* for this purpose. The expression *p(R(F), E)* takes an arbitrary relational algebra expression *E* and returns an instance of a (new) relation called *R*. *R* contains the same tuples as the result of *E* and has the same schema as *E*, but some fields are renamed. The field names in relation *R* are the sarne as in *E*, except for fields renamed in the *renaming list* *F*, which is a list of terms having the form *oldname* $\rightarrow$ *newname* or *position* $\rightarrow$ *newname*. For ρ to be well-defined, references to fields (in the form of *oldnames* or *posit.ions* in the renaming list) may be unarnbiguous and no two fields in the result may have the same name. Sometimes we want to only rename fields or (re)name the relation; we therefore treat both Rand *F* as optional in the use of *p*. (Of course, it is meaningless to omit both.)

For example, the expression *p(C(l → s'id1,5 → sid2),* 81 x *R1)* returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: *C(sidl:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer, *day:* dates).

It is customary to include some additional operators in the algebra, but all of them can be defined in terms of the operators we have defined thus far. (In fact, the renaming operator is needed only for syntactic convenience, and even the n operator is redundant; *R* n 8 can be defined as *R* - *(R* - 8).) We consider these additional operators and their definition in terms of the basic operators in the next two subsections.

4.2.4 Joins

The *join* operation is one of the most useful operations in relational algebra and the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Further, the result of a cross-product is typically much larger than the result of a join, and it is very important to recognize joins and implement them without materializing the underlying cross-product (by applying the selections and projections 'on-the-fly'). For these reasons, joins have received a lot of attention, and there are several variants of the join operation.[1]

Condition Joins

The most general version of the join operation accepts a *join condition* c and a pair of relation instances as arguments and returns a relation instance. The *join cond'it-ion* is identical to a *selection condition* in form. The operation is defined as follows:

$$R \bowtie_c S = \sigma_c(R \times S)$$

Thus ⋈ is defined to be a cross-product followed by a selection. Note that the condition c can (and typically *does)* refer to attributes of both Rand *S*. The reference to an attribute of a relation, say, *R*, can be by positioll (of the form *R.i)* or by Ilame (of the form *R.name)*.

As an example, the result of *Sl* $\bowtie_{S1.sid<R1.sid}$ *R1* is shown in Figure 4.12. Because *sid* appears in both 81 and *R1*, the corresponding fields in the result of the cross-product 81 x *R1* (and therefore in the result of 81 $\bowtie_{S1.sid<R1.sid}$ *R1)*

[1]Several variants of joins are not discussed in this chapter. An important class of joins, called *outer joins*, is discussed in Chapter 5.

are unnamed. Domains are inherited from the corresponding fields of 81 and Rl.

(sid)	*sname*	*rating*	*age*	*(sid)*	*bid*	*day*
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Figure 4.12 51 $\bowtie_{S1.sid<R1.sid}$ R1

Equijoin

A common special case of the join operation $R \bowtie S$ is when the *join condition* consists solely of equalities (connected by $\wedge$) of the form *R.name1 = 8.name2*, that is, equalities between two fields in Rand *S*. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which *8.name2* is dropped. The join operation with this refinement is called **equijoin.**

The schema of the result of an equijoin contains the fields of *R* (with the same names and domains as in *R*) followed by the fields of *S* that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from Rand 8, they are unnamed in the result relation.

We illustrate $S1 \bowtie_{R.sid=S.sid} Rl$ in Figure 4.13. Note that only one field called *sid* appears in the result.

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Figure 4.13 81 $\bowtie_{R.sid=S.sid}$ *HI*

Natural Join

A further special case of the join operation $R \bowtie S$ is an eqUlJom in which equalities arc specified on *all* fields having the same name in Rand *S*. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a *natural jo'in,* and it has the nice property that the result is guaranteed not to have two fields with the saIne name.

The equijoin expression 81 $\bowtie_{R.sid=S.sid}$ *R1* is actually a natural join and can simply be denoted as 81 $\bowtie$ *R1,* since the only common field is *sid.* If the two relations have no attributes in common, 81 $\bowtie$ *Rl* is simply the cross-product.

4.2.5 Division

The division operator is useful for expressing certain kinds of queries for example, "Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise. However, the division operator does not have the same importance as the other operators-it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator).

We discuss division through an example. Consider two relation instances *A* and *B* in which *A* has (exactly) two fields *x* and *y* and *B* has just one field *y,* with the same domain as in *A*. We define the *division* operation *AlB* as the set of all *x* values (in the form of unary tuples) such that for *every y* value in (a tuple of) *B*, there is a tuple *(x,y)* in *A*.

Another way to understand division is as follows. For each *x* value in (the first column of) *A,* consider the set of *y* values that appear in (the second field of) tuples of *A* with that *x* value. If this set contains (all *y* values in) *B*, the *x* value is in the result of *AlB*.

An analogy with integer division may also help to understand division. For integers *A* and *B, AlB* is the largest integer *Q* such that $Q * B \leq A$. :For relation instances *A* and *B, AlB* is the largest relation instance *Q* such that Q x $B \subseteq A$.

Division is illustrated in Figure 4.14. It helps to think of *A* as a relation listing the parts supplied by suppliers and of the *B* relations as listing parts. *AIBi* computes suppliers who supply *all* parts listed in relation instance *Bi*.

Expressing *AlBin* terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all x values in *A* that are not *disqualified.* An *x* value is *disqualified* if by attaching a *y* value from *B*, we obtain a tuple *(x,y)* that is not in *A*. We can compute disqualified tuples using the algebra expression

$$\pi_x((\pi_x(A) \times B) - A)$$

Thus, we can define A/B as

$$\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$

A

sno	pno
81	pI
sl	p2
s1	p3
81	p4
82	pI
s2	p2
83	p2
84	p2
s4	p4

Bl

pno
p2

B2

pno
p2
p4

B3

pno
pl
p2
p4

AlBl

sno
sl
82
83
s4

AlB2

sno
sl
84

AlB3

sno
sl

Figure 4.14 Examples Illustrating Division

To understand the division operation in full generality, we have to consider the case when both *x* and yare replaced by a set of attributes. The generalization is straightforward and left as an exercise for the reader. We discuss two additional examples illustrating division (Queries Q9 and Q10) later in this section.

4.2.6 More Examples of Algebra Queries

We now present several examples to illustrate how to write queries in relational algebra. We use the Sailors, Reserves, and Boats schema for all our examples in this section. We use parentheses as needed to make our algebra expressions unambiguous. Note that all the example queries in this chapter are given a unique query number. The query numbers are kept unique across both this chapter and the SQL query chapter (Chapter 5). This numbering makes it easy to identify a query when it is revisited in the context of relational calculus and SQL and to compare different ways of writing the same query. (All references to a query can be found in the subject index.)

In the rest of this chapter (and in Chapter 5), we illustrate queries using the instances 83 of Sailors, *R2* of Reserves, and *B1* of Boats, shown in Figures 4.15, 4.16, and 4.17, respectively.

(Q1) Find the names of sailors who have rcscT'ucd boat 103.

This query can be written as follows:

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie 8ailoT.5)$$

sid		*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance 83 of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance *R2* of Reserves

We first compute the set of tuples in Reserves with *bid* = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances *R2* and *S3,* it yields a relation that contains just one field, called *sname,* and three tuples *(Dustin), (Horatio),* and *(Lubber).* (Observe that two sailors are called Horatio and only one of them has reserved a red boat.)

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance *HI* of Boats

We can break this query into smaller pieces llsing the renaming operator *p:*

$$p(Temp1, \sigma_{bid=103} ReseTves)$$
$$p(Temp2, Temp1 \bowtie Sailor's)$$
$$\pi_{sname}(Temp2)$$

Notice that because we are only llsing *p* to give names to intermediate relations, the renaming list is optional and is omitted. *TempI* denotes an intermediate relation that identifies reservations of boat 103. *Temp2* is another intermediate relation, and it denotes sailors who have made a reservation in the set *Templ.* The instances of these relations when evaluating this query on the instances *R2* and *S3* are illustrated in Figures 4.18 and 4.19. Finally, we extract the *sname* column from *Temp2.*

sid	*bid*	*day*
22	103	10/8/98
31	103	11/6/98
74	103	9/8/98

Figure 4.18 Instance of *TempI*

sid	*sname*	*rating*	*age*	*bid*	
22	Dustin	7	45.0	103	10/8/98
31	Lubber	8	55.5	103	11/6/98--
74	Horatio	9	35.0	103	9/8/98

Figure 4.19 Instance of *Temp2*

The version of the query using *p* is essentially the same as the original query; the use of *p* is just syntactic sugar. However, there are indeed several distinct ways to write a query in relational algebra. Here is another way to write this query:

Jrsname(CJbid=103(Reserves IX! *Sailors))*

In this version we first compute the natural join of Reserves and Sailors and then apply the selection and the projection.

This example offers a glimpse of the role played by algebra in a relational DBMS. Queries are expressed by users in a language such as SQL. The DBMS translates an SQL query into (an extended form of) relational algebra and then looks for other algebra expressions that produce the same answers but are cheaper to evaluate. If the user's query is first translated into the expression

π_{sname} *(CJbid=103 (Reserves* IX! *Sailors))*

a good query optimizer will find the equivalent expression

π *sname ((CJb·id=103Reserves)* IX! *Sailors)*

Further, the optimizer will recognize that the second expression is likely to be less expensive to compute because the sizes of intermediate relations are smaller, thanks to the early use of selection.

(Q2) Find the names of sailors who ha've reserved a red boat.

π_{sname}(($\sigma_{color='red'}$Boats) IX! *Reserves* $\bowtie$ *SailoI*'s)

This query involves a series of two joins. First, we choose (tuples describing) red boats. Then, we join this set with Reserves (natural join, with equality specified on thè *bid* column) to identify reservations of red boats. Next, we join the resulting intermediate relation with Sailors (natural join, with equality specified on the *sid* column) to retrieve the names of sailors who have rnade reservations for red boats. Finally, we project the sailors' names. The answer, when evaluated on the instances *B1*, *R2*, and *S3*, contains the names Dustin, Horatio, and Lubber.

An equivalent expression is:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'}Boats) \bowtie Reserves) \bowtie Sailors)$$

The reader is invited to rewrite both of these queries by using *p* to make the intermediate relations explicit and compare the schemas of the intermediate relations. The second expression generates intermediate relations with fewer fields (and is therefore likely to result in intermediate relation instances with fewer tuples as well). A relational query optimizer would try to arrive at the second expression if it is given the first.

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'}Sailors) \bowtie Reserves \bowtie Boats)$$

This query is very similar to the query we used to compute sailors who reserved red boats. On instances *Bl, R2,* and *S3,* the query returns the colors green and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$$Jrsname(Sailors \bowtie Reserves)$$

The join of Sailors and Reserves creates an intermediate relation in which tuples consist of a Sailors tuple 'attached to' a Reserves tuple. A Sailors tuple appears in (some tuple of) this intermediate relation only if at least one Reserves tuple has the same *sid* value, that is, the sailor has made some reservation. The answer, when evaluated on the instances *Bl, R2* and *S3,* contains the three tuples $\langle Dustin\rangle$, $\langle HoTatio)$, and $\langle Lubber\rangle$. Even though two sailors called Horatio have reserved a boat, the answer contains only one copy of the tuple $\langle HoTatio)$, because the answer is a *relation,* that is, a *set* of tuples, with no duplicates.

At this point it is worth remarking on how frequently the natural join operation is used in our examples. This frequency is more than just a coincidence based on the set of queries we have chosen to discuss; the natural join is a very natural, widely used operation. In particular, natural join is frequently used when joining two tables on a foreign key field. In Query Q4, for exalnple, the join equates the *sid* fields of Sailors and Reserves, and the *sid* field of Reserves is a foreign key that refers to the *sid* field of Sailors.

(Q5) Find the narnes of sailors who have reserved a Ted OT a gTeen boat.

$$p(Tempboats, (acoloT='rcd'\ Boats) \cup (\sigma_{color='green'}Boats))$$
$$\pi_{sname}(Tempboats \bowtie ReseTves \bowtie Sailors)$$

We identify the set of all boats that are either red or green (Tempboats, which contains boats \vith the *bids* 102, 103, and 104 on instances *E1, R2,* and *S3).* Then we join with Reserves to identify *sids* of sailors who have reserved OIle of these boats; this gives us *sids* 22, 31, 64, and 74 over our example instances. Finally, we join (an intermediate relation containing this set of *sids)* with Sailors to find the names of Sailors with these *sids.* This gives us the names Dustin, Horatio, and Lubber on the instances *B*1, *R2,* and *S3.* Another equivalent definition is the following:

p(Tempboats, (acolor='red'Vcolor='green' Boats))
7fsname(Tempboats $\bowtie$ *Reserves* $\bowtie$ *Sailors)*

Let us now consider a very similar query.

(Q6) Find the names of sailors who have reserved a red and a green boat. It is tempting to try to do this by simply replacing U by n in the definition of Tempboats:

p(Tempboats2, (acolor='red,Eoats) n *(O"color='green,Boats))*
$\pi_{sname}(Tempboats2 \bowtie$ *Reserves* $\bowtie$ *Sailors)*

However, this solution is incorrect-it instead tries to compute sailors who have reserved a boat that is both red and green. (Since *bid* is a key for Boats, a boat can be only one color; this query will always return an empty answer set.) The correct approach is to find sailors who have reserved a red boat, then sailors who have reserved a green boat, and then take the intersection of these two sets:

p(Tempred, π_{sid}*((acolor='red' Eoats)* $\bowtie$ *Reserves))*
p(Tempgreen, $\pi_{sid}((\sigma_{color='green'} Boats)$ $\bowtie$ *Reserves))*
$\pi_{sname}((Tempred$ n *Tempgreen)* $\bowtie$ *Sailors)*

The two temporary relations compute the *sids* of sailors, and their intersection identifies sailors who have reserved both red and green boats. On instances *B*1, *R2,* and *53,* the *sids* of sailors who have reserved a red boat are 22, 31, and 64. The *sids* of sailors who have reserved a green boat are 22, 31, and 74. Thus, sailors 22 and 31 have reserved both a red boat and a green boat; their names are Dustin and Lubber.

This formulation of Query Q6 can easily be adapted to find sailors who have reserved red or green boats (Query Q5); just replace n by U:

$\rho(Tempred, \pi_{sid}((\sigma_{color='red'}$ *Boats)* $\bowtie$ *Reserves))*
p(Tempgreen, $\pi_{sid}($*(O"color='green' Boats)* $\bowtie$ *Reserves))*
$\pi_{sname}((Tempred$ U *Tempgreen)* $\bowtie$ *Sailors)*

In the formulations of Queries Q5 and Q6, the fact that *sid* (the field over which we compute union or intersection) is a key for Sailors is very important. Consider the following attempt to answer Query Q6:

> *p(Tempred, Jrsname((CJcolor='red,Boats)* $\bowtie$ *Reserves* $\bowtie$ *Sailors))*
> *p(Tempgreen,Jrsname((CJcoloT='gTeenlBoats)* $\bowtie$ *Reserves* $\bowtie$ *Sailors))*
> *Tempred* n *Tempgreen*

This attempt is incorrect for a rather subtle reason. Two distinct sailors with the same name, such as Horatio in our example instances, may have reserved red and green boats, respectively. In this case, the name Horatio (incorrectly) is included in the answer even though no one individual called Horatio has reserved a red boat and a green boat. The cause of this error is that *sname* is used to identify sailors (while doing the intersection) in this version of the query, but *sname* is not a key.

(Q7) Find the names of sailors who have reser-ved at least two boats.

> *p(Reser-vations,* $\pi_{sid,sname,bid}$*(Sailors* $\bowtie$ *Reserves))*
> *p(Reservationpairs(l* $\rightarrow$ *sid1,* 2 $\rightarrow$ *sname1,* 3 $\rightarrow$ *bid*1, 4 $\rightarrow$ *sid*2,
> 5 $\rightarrow$ *sname2,* 6 $\rightarrow$ *bid2), Reservations* x *Reservations)*
> π_{sname1} CJ$_{(sid1=sid2)\wedge(bid1\neq bid2)}$ *Reservationpair-s*

First, we compute tuples of the form *(sid,sname, bid)*, where sailor *sid* has made a reservation for boat *bid;* this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: To show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances *B*1, *R2,* and *S3,* each of the sailors with *sids* 22, 31, and 64 have reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names Dustin, Horatio, and Lubber.

Notice that we included *sid* in Reservations because it is the key field identifying sailors, and we need it to check that two Reservations tuples involve the same sailor. As noted in the previous example, we cannot use *sname* for this purpose.

(Q8) Find the sids of sailors with age over 20 *who have not TeseTved a Ted boat.*

> $\pi_{sid}(\sigma_{age>20}Sailors)$ –
> *7rsid((CJ*$_{co}$*[oT='red,Boats)* $\bowtie$ *Reserves* $\bowtie$ *Sailors*)

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors. We first identify sailors aged over 20 (over instances $B1$, $R2$, and $S3$, *sids* 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have reserved a red boat (*sids* 22, 31, and 64), to obtain the answer (*sids* 29, 32, 58, 74, 85, and 95). If we want to compute the names of such sailors, we must first compute their *sids* (as shown earlier) and then join with Sailors and project the *sname* values.

(Q9) Find the names of sailors who have reserved all boats.

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The intermediate relation Tempsids is defined using division and computes the set of *sids* of sailors who have reserved every boat (over instances $B1$, $R2$, and $S3$, this is just *sid* 22). Note how we define the two relations that the division operator (/) is applied to—the first relation has the schema *(sid,bid)* and the second has the schema *(bid)*. Division then returns all *sids* such that there is a tuple *(sid,bid)* in the first relation for each *bid* in the second. Joining Tempsids with Sailors is necessary to associate names with the selected *sids;* for sailor 22, the name is Dustin.

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute *bids* only of boats named *Interlake* in defining the second argument to the division operator. Over instances $B1$, $R2$, and $S3$, Tempsids evaluates to *sids* 22 and 64, and the answer contains their names, Dustin and Horatio.

4.3 RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus we present in detail is called the **tuple relational calculus** (TRC). Variables in TRC take on tuples as values. In another variant, called the **domain relational calculus (DRC)**, the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE. We discuss DRC in Section 4.3.2.[2]

4.3.1 Tuple Relational Calculus

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form $\{ T \mid p(T) \}$, where T is a tuple variable and $p(T)$ denotes a *formula* that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula $p(T)$ evaluates to **true** with $T = t$. The language for writing formulas $p(T)$ is thus at the heart of TRC and essentially a simple subset of *first-order logic*. As a simple example, consider the following query.

(*Q*11) *Find all sailors with a rating above 7.*

$$\{S \mid S \in Sailors \land S.rating > 7\}$$

When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test $S.rating > 7$ is applied. The answer contains those instances of S that pass this test. On instance *S3* of Sailors, the answer contains Sailors tuples with *sid* 31, 32, 58, 71, and 74.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. Let *Rel* be a relation name, R and S be tuple variables, a be an attribute of R, and b be an attribute of S. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$. An **atomic formula** is one of the following:

- $R \in Rel$
- $R.a$ op $S.b$
- $R.a$ op *constant*, or *constant* op $R.a$

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas and $p(R)$ denotes a formula in which the variable R appears:

2The material on DRC is referred to in the (online) chapter on QBE; with the exception of this chapter, the material on DRC and TRC can be omitted without loss of continuity.

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

In the last two clauses, the quantifiers $\exists$ and $\forall$ are said to bind the variable R. A variable is said to be free in a formula or *subformuia* (a formula contained in a larger formula) if the (sub)formula does not contain an occurrence of a quantifier that binds it.[3]

We observe that every variable in a TRC formula appears in a subformula that is atomic, and every relation schema specifies a domain for each field; this observation ensures that each variable in a TRC formula has a well-defined domain from which values for the variable are drawn. That is, each variable has a well-defined *type,* in the programming language sense. Informally, an atomic formula $R \in Rei$ gives R the type of tuples in *Rel,* and comparisons such as $R.a$ op $S.b$ and $R.a$ op *constant* induce type restrictions on the field $R.a$. If a variable R does not appear in an atomic formula of the form $R \in Rei$ (Le., it appears only in atomic formulas that are comparisons), we follow the convention that the type of R is a tuple whose fields include all (and only) fields of R that appear in the formula.

We do not define types of variables formally, but the type of a variable should be clear in most cases, and the important point to note is that comparisons of values having different types should always fail. (In discussions of relational calculus, the simplifying assumption is often made that there is a single domain of constants and this is the domain associated with each field of each relation.)

A TRC query is defined to be expression of the form $\{T \mid p(T)\}$, where T is the only free variable in the formula p.

Semantics of TRC Queries

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The answer to a TRC query $\{T \mid p(T)\}$, as noted earlier, is the set of all tuples t for which the formula $peT)$ evaluates to true with variable T assigned the tuple value t. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to true.

[3]We make the assumption that each variable in a formula is either free or bound by exactly one occurrence of a quantifier, to avoid worrying about details such as nested occurrences of quantifiers that bind some, but not all, occurrences of variables.

A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, F evaluates to (or simply 'is') true if one of the following holds:

- F is an atomic formula $R \in Rel$, and R is assigned a tuple in the instance of relation *Rel*.
- F is a comparison $R.a$ op $S.b$, $R.a$ op *constant*, or *constant* op $R.a$, and the tuples assigned to Rand S have field values $R.a$ and $S.b$ that make the comparison true.
- F is of the form $\neg p$ and p is not true, or of the form $p \wedge q$, and both p and q are true, or of the form $p \vee q$ and one of them is true, or of the form $p \Rightarrow q$ and q is true whenever[4] p is true.
- F is of the form $\exists R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$, including the variable R,[5] that makes the formula $p(R)$ true.
- F is of the form $\forall R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$ that makes the formula $p(R)$ true no matter what tuple is assigned to R.

Examples of TRC Queries

We now illustrate the calculus through several examples, using the instances $B1$ of Boats, $R2$ of Reserves, and $S3$ of Sailors shown in Figures 4.15, 4.16, and 4.17. We use parentheses as needed to make our formulas unambiguous. Often, a formula $p(R)$ includes a condition $R \in Rel$, and the meaning of the phrases *some tuple R* and *for all tuples R* is intuitive. We use the notation $\exists R \in Rel(p(R))$ for $\exists R(R \in Rel \wedge p(R))$. Similarly, we use the notation $\forall R \in Rel(p(R))$ for $\forall R(R \in Rel \Rightarrow p(R))$.

(*Q*12) *Find the names and ages of sailors with a rating above 7.*

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge Pname = S.sname \wedge Page = S.age)\}$$

This query illustrates a useful convention: P is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of P mentioned and P does not range over any of the relations in the query; that is, there is no subformula of the form $P \in Relname$. The result of this query is a relation with two fields, *name* and *age*. The atomic

[4] *WheneveT* should be read more precisely as 'for all assignments of tuples to the free variables.'
[5]Note that some of the free variables in $p(R)$ (e.g., the variable R itself) IIlay be bound in P.

formulas *P.name* = *S.sname* and *Page* = *S.age* give values to the fields of an answer tuple *P*. On instances *E1*, *R2*, and *S3*, the answer is the set of tuples ⟨*Lubber*, 55.5⟩, ⟨*Andy*, 25.5), ⟨*Rusty*, 35.0⟩, *(Zorba*, 16.0), and ⟨*Horatio*, *35.0).*

(Q1S) Find the sailor name, boat'id, and reservation date for each reservation.

{P I ∃*R* E *Reserves* ∃*S* E *Sailors*

(R.sid = *8.sid!\ P.bid* = *R.bid!\ P.day* = *R.day* !\ *P.sname* = *S.sname)}*

For each Reserves tuple, we look for a tuple in Sailors with the same *sid*. Given a pair of such tuples, we construct an answer tuple *P* with fields *sname, bid,* and *day* by copying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple. The answer to this query on instances *E1, R2,* and 83 is shown in Figure 4.20.

sname	*bid*	*day*
Dustin	101	10/10/98
Dustin	102	10/10/98
Dustin	103	10/8/98
Dustin	104	10/7/98
Lubber	102	11/10/98
Lubber	103	11/6/98
Lubber	104	11/12/98
Horatio	101	9/5/98
Horatio	102	9/8/98
Horatio	103	9/8/98

Figure 4.20 Answer to Query Q13

(*Q*1) *Find the names of sailors who have reserved boat 103.*

{P I ∃*S* E *Sailors* ∃*R* E *Reserves(R.s'id* = *S.sid!\ R.b'id* = 103 *!\Psname* = *8.snarne)}*

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves having the same value in the *sid* field and with *b'id* = 103." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple *P* contains just one field, *sname*.

(Q2) Find the names of sailors who have reserved a red boat.

{P I ∃*S* E *Sailors* ∃*R* E *Reserves(R.sid* = *5.sid* !\ *P.sname* = *S.8name*

$\wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that $S.sid = R.sid$, $R.bid = B.bid$, and $B.color = 'red'$." Another way to write this query, which corresponds more closely to this reading, is as follows:

$$\{P \mid \exists S \in Sailors\ \exists R \in Reserves\ \exists B \in Boats$$
$$(R.sid = S.sid \wedge B.bid = R.bid \wedge B.color = 'red' \wedge P.sname = S.sname)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{P \mid \exists S \in Sailors\ \exists R1 \in Reserves\ \exists R2 \in Reserves$$
$$(S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid$$
$$\wedge P.sname = S.sname)\}$$

Contrast this query with the algebra version and see how much simpler the calculus version is. In part, this difference is due to the cumbersome renaming of fields in the algebra version, but the calculus version really is simpler.

(Q9) Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors\ \forall B \in Boats$$
$$(\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

This query was expressed using the division operator in relational algebra. Note how easily it is expressed in the calculus. The calculus query directly reflects how we might express the query in English: "Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B."

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply that the sailor has reserved it. Observe that since we can return an entire sailor tuple as the answer instead of just the sailor's name, we avoided introducing a new free variable (e.g., the variable P in the previous example) to hold the answer values. On instances $B1$, $R2$, and $S3$, the answer contains the Sailors tuples with *sids* 22 and 31.

We can write this query without using implication, by observing that an expression of the form $p \Rightarrow q$ is logically equivalent to $\neg p \vee q$:

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$

$$(B.color \neq 'red' \vee (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: "Find sailors S such that, for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B."

4.3.2 Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form $\{\langle x_1, x_2, \ldots, x_n \rangle \mid p(\langle x_1, x_2, \ldots, x_n \rangle)\}$, where each x_i is either a *domain variable* or a constant and $p(\langle x_1, x_2, \ldots, x_n \rangle)$ denotes a **DRC formula** whose only free variables are the variables among the x_i, $1 \leq i \leq n$. The result of this query is the set of all tuples $\langle x_1, x_2, \ldots, x_n \rangle$ for which the formula evaluates to true.

A DRC formula is defined in a manner very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$ and let X and Y be domain variables. An **atomic formula** in DRC is one of the following:

- $\langle x_1, x_2, \ldots, x_n \rangle \in Rel$, where Rel is a relation with n attributes; each x_i, $1 \leq i \leq n$ is either a variable or a constant
- X op Y
- X op *constant*, or *constant* op X

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas and $p(X)$ denotes a formula in which the variable X appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

The reader is invited to compare this definition with the definition of TRC formulas and see how closely these two definitions correspond. We will not define the semantics of DRC formulas formally; this is left as an exercise for the reader.

Examples of DRC Queries

We now illustrate DRC through several examples. The reader is invited to compare these with the TRC versions.

(*Q*11) *Find all sailors with a rating above 7.*

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

This differs from the TRC version in giving each attribute a (variable) name. The condition $\langle I, N, T, A\rangle \in Sailors$ ensures that the domain variables I, N, T, and A are restricted to be fields of the *same* tuple. In comparison with the TRC query, we can say $T > 7$ instead of $S.rating > 7$, but we must specify the tuple $\langle I, N, T, A\rangle$ in the result, rather than just S.

(*Q*1) *Find the names of sailors who have reserved boat 103.*

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors$$
$$\wedge \exists Ir, Br, D(\langle 11', Br, D\rangle \in Reserves \wedge 11' = I \wedge Br = 103))\}$$

Note that only the *sname* field is retained in the answer and that only N is a free variable. We use the notation $\exists Ir, Br, D(\ldots)$ as a shorthand for $\exists Ir(\exists Br(\exists D(\ldots)))$. Very often, all the quantified variables appear in a single relation, as in this example. An even more compact notation in this case is $\exists\langle Ir, Br, D\rangle \in Reserves$. With this notation, which we use henceforth, the query would be as follows:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors$$
$$\wedge \exists\langle Ir, Br, D\rangle \in Reserves(Ir = I \wedge Br = 103))\}$$

The comparison with the corresponding TRC formula should now be straightforward. This query can also be written as follows; note the repetition of variable I and the use of the constant 103:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors$$
$$\wedge \exists D(\langle 1, 103, D\rangle \in Reserves))\}$$

(*Q2*) *Find the names of sailors who have Teserved a red boat.*

$$\{\langle N\rangle \mid \exists I, T, A(\langle 1, N, T, A\rangle \in Sailors$$
$$\wedge \exists\langle I, Br, D\rangle \in ReseTves \wedge \exists\langle Br, BN, 'Ted'\rangle \in Boats)\}$$

(*Q7*) *Find the names of sailors who have TeseTved at least two boats.*

$$\{\langle N\rangle \mid \exists I, T, A(\langle 1, N, T, A\rangle \in Sailors \wedge$$
$$\exists Br1, BT2, Dl, D2(\langle 1, Brl, DI\rangle \in Reserves$$
$$\wedge\langle 1, Br2, D2\rangle \in Reserves \wedge Brl \neq Br2))\}$$

Note how the repeated use of variable *I* ensures that the same sailor has reserved both the boats in question.

(Q9) Find the names of sailors who have Teserved all boat8.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge$$
$$\forall B, BN, C(\neg(\langle B, BN, C\rangle \in Boats) \vee$$
$$(\exists\langle Ir, Br, D\rangle \in Reserves(I = IT \wedge BT = B))))\}$$

This query can be read as follows: "Find all values of *N* such that some tuple *(I, N, T, A)* in Sailors satisfies the following condition: For every $\langle B, BN, C\rangle$, either this is not a tuple in Boats or there is some tuple *(IT, BT, D)* in Reserves that proves that Sailor *I* has reserved boat *B*." The $\forall$ quantifier allows the domain variables *B*, *BN*, and C to range over all values in their respective attribute domains, and the pattern '$\neg(\langle B, BN, C\rangle \in Boats) \vee$' is necessary to restrict attention to those values that appear in tuples of Boats. This pattern is common in DRC formulas, and the notation $\forall\langle B, BN, C\rangle \in Boats$ can be used as a shortcut instead. This is similar to the notation introduced earlier for $\exists$. With this notation, the query would be written as follows:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \forall\langle B, BN, C\rangle \in Boats$$
$$(\exists\langle Ir, BT, D\rangle \in ReseTves(I = IT \wedge BT = B)))\}$$

(Q14) Find sailoTs who have TeseTved all Ted boats.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in SailoTs \wedge \forall\langle B, BN, C\rangle \in Boats$$
$$(C = 'red' \Rightarrow \exists\langle Ir, BT, D\rangle \in Reserves(I = IT \wedge Br = B))\}$$

Here, we find all sailors such that, for every red boat, there is a tuple in Reserves that shows the sailor has reserved it.

4.4 EXPRESSIVE POWER OF ALGEBRA AND CALCULUS

We presented two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can. Can every query that can be expressed in relational calculus also be expressed in relational algebra? Before we answer this question, we consider a major problem with the calculus as we presented it.

Consider the query $\{S \mid \neg(S \in Sailors)\}$. This query is syntactically correct. However, it asks for all tuples *S* such that *S* is not in (the given instance of)

Sailors. The set of such S tuples is obviously infinite, in the context of infinite domains such as the set of all integers. This simple example illustrates an *unsafe* query. It is desirable to restrict relational calculus to disallow unsafe queries.

We now sketch how calculus queries are restricted to be safe. Consider a set I of relation instances, with one instance per relation that appears in the query Q. Let $Dom(Q, 1)$ be the set of all constants that appear in these relation instances I or in the formulation of the query Q itself. Since we allow only finite instances I, $Dom(Q, 1)$ is also finite.

For a calculus formula Q to be considered safe, at a minimum we want to ensure that, for any given I, the set of answers for Q contains only values in $Dom(Q, 1)$. While this restriction is obviously required, it is not enough. Not only do we want the set of answers to be composed of constants in $Dom(Q, 1)$, we wish to *compute* the set of answers by examining only tuples that contain constants in $Dom(Q, 1)$! This wish leads to a subtle point associated with the use of quantifiers $\forall$ and $\exists$: Given a TRC formula of the form $\exists R(p(R))$, we want to find all values for variable R that make this formula `true` by checking only tuples that contain constants in $Dom(Q, 1)$. Similarly, given a TRC formula of the form $\forall R(p(R))$, we want to find any values for variable R that make this formula `false` by checking only tuples that contain constants in $Dom(Q, 1)$.

We therefore define a *safe* TRC formula Q to be a formula such that:

1. For any given I, the set of answers for Q contains only values that are in $Dom(Q, 1)$.
2. For each subexpression of the form $\exists R(p(R))$ in Q, if a tuple r (assigned to variable R) makes the formula `true`, then r contains only constants in $Dorn(Q,I)$.
3. For each subexpression of the form $\forall R(p(R))$ in Q, if a tuple r (assigned to variable R) contains a constant that is not in $Dom(Q, 1)$, then r must make the formula `true`.

Note that this definition is not *constructive*, that is, it does not tell us how to check if a query is safe.

The query $Q = \{S \mid \neg(S \in Sailors)\}$ is unsafe by this definition. $Dom(Q,1)$ is the set of all values that appear in (an instance I of) Sailors. Consider the instance Sl shown in Figure 4.1. The answer to this query obviously includes values that do not appear in $Dorn(Q,81)$.

Returning to the question of expressiveness, we can show that every query that can be expressed using a *safe* relational calculus query can also be expressed as a relational algebra query. The expressive power of relational algebra is often used as a metric of how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be **relationally complete.** A practical query language is expected to be relationally complete; in addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

4.5 REVIEW QUESTIONS

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

- What is the input to a relational query? What is the result of evaluating a query? **(Section 4.1)**
- Database systems use some variant of relational algebra to represent query evaluation plans. Explain why algebra is suitable for this purpose. **(Section 4.2)**
- Describe the selection operator. What can you say about the cardinality of the input and output tables for this operator? (That is, if the input has k tuples, what can you say about the output?) Describe the projection operator. What can you say about the cardinality of the input and output tables for this operator? **(Section 4.2.1)**
- Describe the set operations of relational algebra, including union (U), intersection (n), set-difference (-), and cross-product (x). For each, what can you say about the cardinality of their input and output tables? **(Section 4.2.2)**
- Explain how the renaming operator is used. Is it required? That is, if this operator is not allowed, is there any query that can no longer be expressed in algebra? **(Section 4.2.3)**
- Define all the variations of the join operation. Why is the join operation given special attention? Cannot we express every join operation in terms of cross-product, selection, and projection? **(Section 4.2.4)**
- Define the division operation in terms of the basic relational algebra operations. Describe a typical query that calls for division. Unlike join, the division operator is not given special treatment in database systems. Explain why. **(Section 4.2.5)**

- Relational calculus is said to be a *declarative* language, in contrast to algebra, which is a *procedural* language. Explain the distinction. **(Section 4.3)**
- How does a relational calculus query 'describe' result tuples? Discuss the subset of first-order predicate logic used in tuple relational calculus, with particular attention to universal and existential quantifiers, bound and free variables, and restrictions on the query formula. **(Section 4.3.1).**
- What is the difference between tuple relational calculus and domain relational calculus? **(Section 4.3.2).**
- What is an *unsafe* calculus query? Why is it important to avoid such queries? **(Section 4.4)**
- Relational algebra and relational calculus are said to be equivalent in expressive power. Explain what this means, and how it is related to the notion of *relational completeness.* **(Section 4.4)**

EXERCISES

Exercise 4.1 Explain the statement that relational algebra operators can be *composed.* Why is the ability to compose operators important?

Exercise 4.2 Given two relations *R1* and *R2,* where *R1* contains N1 tuples, *R2* contains N2 tuples, and N2 > N1 > 0, give the minimum and maximum possible sizes (in tuples) for the resulting relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for *R1* and *R2* needed to make the expression meaningful:

(1) $R1 \cup R2$, (2) $R1 \cap R2$, (3) $R1 - R2$, (4) $R1 \times R2$, (5) $\sigma_{a=5}(R1)$, (6) $\pi_a(R1)$, and (7) $R1/R2$

Exercise 4.3 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 key fields are underlined, and the domain of each field is listed after the field name. Therefore *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus:

1. Find the *names* of suppliers who supply some red part.
2. Find the *sids* of suppliers who supply some red or green part.
3. Find the *sids* of suppliers who supply some red part or are at 221 Packer Ave.
4. Find the *sids* of suppliers who supply some rcd part and some green part.

5. Find the *sids* of suppliers who supply every part.
6. Find the *sids* of suppliers who supply every red part.
7. Find the *sids* of suppliers who supply every red or green part.
8. Find the *sids* of suppliers who supply every red part or supply every green part.
9. Find pairs of *sids* such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.
10. Find the *pids* of parts supplied by at least two different suppliers.
11. Find the *pids* of the most expensive parts supplied by suppliers named Yosemite Sham.
12. Find the *pids* of parts supplied by every supplier at less than $200. (If any supplier either does not supply the part or charges more than $200 for it, the part is not selected.)

Exercise 4.4 Consider the Supplier-Parts-Catalog schema from the previous question. State what the following queries compute:

1. $\pi_{sname}(\pi_{sid}(\sigma_{color='red'}$ *Parts)* !XI *(O'cost<lOoCatalog)* !XI *Suppliers)*
2. π*sname* $(\pi_{sid}((\sigma_{\text{color='red'}}$ *Parts)* !XI $(\sigma_{cost<}$*looCatalog)* !XI $Suppliers))$
3. $(\pi_{sname}$*((O'color'='red' Parts)* $\bowtie$ *(crcost<looCatalog)* !XI *Suppl'iers))* n

   $(\pi_{sname}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$

4. *(1fsid((crcolor='red,Parts)* !XI *(crcost<10oCatalog)* $\bowtie$ *Suppliers))* n

   $(\pi_{sid}((\sigma_{color='green'}$ *Parts)* $\bowtie$ *(crcost<lOoCatalog)* !XI *Suppliers))*

5. $\pi_{sname}((\pi_{sid,sname}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap$

   $(\pi_{sid,sname}(($*σCO!07'='green' Parts)* !XI *(σcost< lOoCatalog)* $\bowtie$ *Suppliers)))*

Exercise 4.5 Consider the following relations containing airline flight information:

> <u>Flights(fino:</u> <u>integer,</u> *from:* string, *to:* string,
> *d·istance:* integer, *departs:* time, *arrives:* time)
> Aircraft(<u>*aid:*</u> <u>integer,</u> *aname:* string, *cTuisingrange:* integer)
> Certified(*eid:* <u>integer,</u> <u>*aid:*</u> <u>integer)</u>
> Employees(<u>*eid:*</u> <u>integer,</u> *ename:* string, *salary:* integer)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft (otherwise, he or she would not qualify as a pilot), and only pilots are certified to fly.

Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus. Note that some of these queries may not be expressible in relational algebra (and, therefore, also not expressible in tuple and domain relational calculus)! For such queries, informally explain why they cannot be expressed. (See the exercises at the end of Chapter 5 for additional queries over the airline schenla.)

1. Finel the *eids* of pilots certified for some Boeing aircraft.
2. Find the *names* of pilots certified for some Boeing aircraft.
3. Find the *aids* of all aircraft that. can be used on non-stop flights from Bonn to Madras.

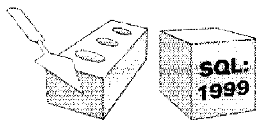
4. Identify the flights that can be piloted by every pilot whose salary is more than \$100,000.
5. Find the names of pilots who can operate planes with a range greater than 3,000 miles but are not certified on any Boeing aircraft.
6. Find the *eids* of employees who make the highest salary.
7. Find the *eids* of employees who make the second highest salary.
8. Find the *eids* of employees who are certified for the largest number of aircraft.
9. Find the *eids* of employees who are certified for exactly three aircraft.
10. Find the total amount paid to employees as salaries.
11. Is there a sequence of flights from Madison to Timbuktu? Each flight in the sequence is required to depart from the city that is the destination of the previous flight; the first flight must leave Madison, the last flight must reach Timbuktu, and there is no restriction on the number of intermediate flights. Your query must determine whether a sequence of flights from Madison to Timbuktu exists for *any* input Flights relation instance.

Exercise **4.6** What is *relational completeness?* **If** a query language is relationally complete, can you write any desired query in that language?

Exercise **4.7** What is an *unsafe* query? Give an example and explain why it is important to disallow such queries.

BIBLIOGRAPHIC NOTES

Relational algebra was proposed by Codd in [187], and he showed the equivalence of relational algebra and TRC in [189]. Earlier, Kuhns [454] considered the use of logic to pose queries. LaCroix and Pirotte discussed DRC in [459]. Klug generalized the algebra and calculus to include aggregate operations in [439]. Extensions of the algebra and calculus to deal with aggregate functions are also discussed in [578]. Merrett proposed an extended relational algebra with quantifiers such as *the number of* that go beyond just universal and existential quantification [530]. Such generalized quantifiers are discussed at length in [52].



5

SQL: QUERIES, CONSTRAINTS, TRIGGERS

- What is included in the SQL language? What is SQL:1999?
- How are queries expressed in SQL? How is the meaning of a query specified in the SQL standard?
- How does SQL build on and extend relational algebra and calculus?
- What is grouping? How is it used with aggregate operations?
- What are nested queries?
- What are *null* values?
- How can we use queries in writing complex integrity constraints?
- What are triggers, and why are they useful? How are they related to integrity constraints?
- Key concepts: SQL queries, connection to relational algebra and calculus; features beyond algebra, DISTINCT clause and multiset semantics, grouping and aggregation; nested queries, correlation; set-comparison operators; *null* values, outer joins; integrity constraints specified using queries; triggers and active databases, event-condition-action rules.

What men or gods are these? What Inaiclens loth?
What mad pursuit? What struggle to escape?
\Vhat pipes and tilubrels? \Vhat wild ecstasy?

— John Keats, *Odc on a Grecian Urn*

Structured Query Language (SQL) is the most widely used conunercial relational database language. It was originally developed at IBlVI in the SEQUEL-

SQL Standards Conformance: SQL:1999 has a collection of features called Core SQL that a vendor must implement to claim conformance with the SQL:1999 standard. It is estimated that all the major vendors can comply with Core SQL with little effort. Many of the remaining features are organized into packages.

For example, packages address each of the following (with relevant chapters in parentheses): *enhanced date and time, enhanced integrity management and active databases* (this chapter), *external language 'interfaces* (Chapter 6), *OLAP* (Chapter 25), and *object features* (Chapter 23). The SQL/MM standard complements SQL:1999 by defining additional packages that support *data mining* (Chapter 26), *spatial data* (Chapter 28) and *text documents* (Chapter 27). Support for XML data and queries is forthcoming.

XRM and System-R projects (1974-1977). Almost immediately, other vendors introduced DBMS products based on SQL, and it is now a de facto standard. SQL continues to evolve in response to changing needs in the database area. The current ANSI/ISO standard for SQL is called SQL:1999. While not all DBMS products support the full SQL:1999 standard yet, vendors are working toward this goal and most products already support the core features. The SQL:1999 standard is very close to the previous standard, SQL-92, with respect to the features discussed in this chapter. Our presentation is consistent with both SQL-92 and SQL:1999, and we explicitly note any aspects that differ in the two versions of the standard.

5.1 OVERVIEW

The SQL language has several aspects to it.

- The Data Manipulation Language (DML): This subset of SQL allows users to pose queries and to insert, delete, and modify rows. Queries are the main focus of this chapter. We covered DML commands to insert, delete, and modify rows in Chapter 3.
- The Data Definition Language (DDL): This subset of SQL supports the creation, deletion, and modification of definitions for tables and views. *Integrity constraints* can be defined on tables, either when the table is created or later. We cocvered the DDL features of SQL in Chapter 3. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes.
- Triggers and Advanced Integrity Constraints: The new SQL:1999 standard includes support for *triggers*, which are actions executed by the

DBMS whenever changes to the database meet conditions specified in the trigger. We cover triggers in this chapter. SQL allows the use of queries to specify complex integrity constraint specifications. We also discuss such constraints in this chapter.

- Embedded and Dynamic SQL: Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dynamic SQL features allow a query to be constructed (and executed) at run-time. \Ve cover these features in Chapter 6.
- Client-Server Execution and Remote Database Access: These commands control how a *client* application program can connect to an SQL database *server*, or access data from a database over a network. We cover these commands in Chapter 7.
- Transaction Management: Various commands allow a user to explicitly control aspects of how a transaction is to be executed. We cover these commands in Chapter 21.
- Security: SQL provides mechanisms to control users' access to data objects such as tables and views. We cover these in Chapter 2l.
- Advanced features: The SQL:1999 standard includes object-oriented features (Chapter 23), recursive queries (Chapter 24), decision support queries (Chapter 25), and also addresses emerging areas such as data mining (Chapter 26), spatial data (Chapter 28), and text and XML data management (Chapter 27).

5.1.1 Chapter Organization

The rest of this chapter is organized as follows. We present basic SQL queries in Section 5.2 and introduce SQL's set operators in Section 5.3. We discuss nested queries, in which a relation referred to in the query is itself defined within the query, in Section 5.4. We cover aggregate operators, which allow us to write SQL queries that are not expressible in relational algebra, in Section 5.5. \Ve discuss *null* values, which are special values used to indicate unknown or nonexistent field values, in Section 5.6. We discuss complex integrity constraints that can be specified using the SQL DDL in Section 5.7, extending the SQL DDL discussion from Chapter 3; the new constraint specifications allow us to fully utilize the query language capabilities of SQL.

Finally, we discuss the concept of an *active database* in Sections 5.8 and 5.9. An active database has a collection of triggers, which are specified by the DBA. A trigger describes actions to be taken when certain situations arise. The DBMS lllonitors the database, detects these situations, and invokes the trigger.

The SQL:1999 standard requires support for triggers, and several relational DBMS products already support some form of triggers.

About the Examples

We will present a number of sample queries using the following table definitions:

Sailors(*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
Boats(*bid:* integer, *bname:* string, *color:* string)
Reserves(*sid:* integer, *bid:* integer, *day:* date)

We give each query a unique number, continuing with the numbering scheme used in Chapter 4. The first new query in this chapter has number Q15. Queries Q1 through Q14 were introduced in Chapter 4.[1] We illustrate queries using the instances 83 of Sailors, *R2* of Reserves, and *B1* of Boats introduced in Chapter 4, which we reproduce in Figures 5.1, 5.2, and 5.3, respectively.

All the example tables and queries that appear in this chapter are available online on the book's webpage at

http://www.cs.wisc.edu/-dbbook

The online material includes instructions on how to set up Orade, IBM DB2, Microsoft SQL Server, and MySQL, and scripts for creating the example tables and queries.

5.2 THE FORM OF A BASIC SQL QUERY

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual evaluation strategy.* A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The basic form of an SQL query is as follows:

```
SELECT  [DISTINCT] select-list
FROM    from-list
WHERE   qualification
```

[1]All references to a query can be found in the subject index for the book.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance 53 of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance *R2* of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance *BI* of Boats

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. The close relationship between SQL and relational algebra is the basis for query optimization in a relational DBMS, as we will see in Chapters 12 and 15. Indeed, execution plans for SQL queries are represented using a variation of relational algebra expressions (Section 15.1).

Let us consider a simple example.

(Q15) Find the' names and ages of all sailors.

```
SELECT  DISTINCT  S.sname, S.age
FROM    Sailors S
```

The answer is a *set* of rows, each of which is a pair *(sname, age)*. If two or more sailors have the same name and age, the answer still contains just one pair with that name and age. This query is equivalent to applying the projection operator of relational algebra.

If we omit the keyword DISTINCT, we would get a copy of the row *(s,a)* for each sailor with name *s* and age *a;* the answer would be a *multiset* of rows. A **multiset** is similar to a set in that it is an unordered collection of elements, but there could be several copies of each element, and the number of copies is significant-two multisets could have the same elements and yet be different because the number of copies is different for some elements. For example, {a, b, b} and {b, a, b} denote the same multiset, and differ from the multiset {a, a, b}.

The answer to this query with and without the keyword DISTINCT on instance 53 of Sailors is shown in Figures 5.4 and 5.5. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35.

snarne	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Art	25.5
Bob	63.5

Figure 5.4 Answer to Q15

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Figure 5.5 Answer to Q15 without DISTINCT

Our next query is equivalent to an application of the selection operator of relational algebra.

(*Q*11) *Find all sailors with a rating above 7.*

```
SELECT  S.sid, S.sname, S.rating, S.age
FROM    Sailors AS S
WHERE   S.rating > 7
```

This query uses the optional keyword AS to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides a convenient shorthand: We can simply write SELECT *. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained because the schema of the result is not clear from the query itself; we have to refer to the schema of the underlying Sailors table.

As these two examples illustrate, the SELECT clause is actually used to do *projection,* whereas *selections* in the relational algebra sense are expressed using the WHERE clause! This mismatch between the naming of the selection and projection operators in relational algebra and the syntax of SQL is an unfortunate historical accident.

We now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form *expression* op *expression,* where op is one of the comparison operators {<, <=, =, <>, >=, >}.[2] An *expression* is a *column* name, a *constant,* or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates,* that is, two copies of the same row. The default is that duplicates are not eliminated.

Although the preceding rules describe (informally) the syntax of a basic SQL query, they do not tell us the *meaning* of a query. The answer to a query is itself a relation which is a *multiset* of rows in SQL!--whose contents can be understood by considering the following conceptual evaluation strategy:

1. Cmnpute the cross-product of the tables in the from-list.
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

[2]ExpressiollS with NOT can always be replaced by equivalent expressions without NOT given the set of comparison operators just listed.

This straightforward conceptual evaluation strategy makes explicit the rows that must be present in the answer to the query. However, it is likely to be quite inefficient. We will consider how a DB:MS actually evaluates queries in later chapters; for now, our purpose is simply to explain the meaning of a query. \Ve illustrate the conceptual evaluation strategy using the following query':

(Q1) *Find the names of sailors 'Who have reseTved boat number 103.*

It can be expressed in SQL as follows.

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid = R.sid AND R.bid=103
```

Let us compute the answer to this query on the instances *R3* of Reserves and 84 of Sailors shown in Figures 5.6 and 5.7, since the computation on our usual example instances *(R2* and 83) would be unnecessarily tedious.

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 5.6 Instance *R3* of Reserves

sid	*sname*	*Tating*	*age*
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 5.7 Instance 54 of Sailors

The first step is to construct the cross-product 84 x *R3,* which is shown in Figure 5.8.

sid	*sname*	*rating*	*age*	*sid*	*bid*	*day*
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	3.5.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Figure 5.8 *S4* x *R3*

The second step is to apply the qualification *S.sid* = *R.sid* AND *R.bid=103.* (Note that the first part of this qualification requires a join operation.) This step eliminates all but the last row from the instance shown in Figure 5.8. The third step is to eliminate unwanted columns; only *sname* appears in the SELECT clause. This step leaves us with the result shown in Figure .5.9, which is a table with a single column and, as it happens, just one row.

sname
rusty

Figure 5.9 Answer to Query Ql 011 R3 and 84

5.2.1 Examples of Basic SQL Queries

We now present several example queries, many of which were expressed earlier in relational algebra and calculus (Chapter 4). Our first example illustrates that the use of range variables is optional, unless they are needed to resolve an ambiguity. Query Ql, which we discussed in the previous section, can also be expressed as follows:

```
SELECT  sname
FROM    Sailors 5, Reserves R
WHERE   S.sid = R.sid AND bid=103
```

Only the occurrences of *sid* have to be qualified, since this column appears in both the Sailors and Reserves tables. An equivalent way to write this query is:

```
SELECT  SHame
FROM    Sailors, Reserves
WHERE   Sailors.sid = Reserves.sid AND bid=103
```

This query shows that table names can be used implicitly as row variables. Range variables need to be introduced explicitly only when the FROM clause contains more than one occurrence of a relation.[3] However, we recommend the explicit use of range variables and full qualification of all occurrences of columns with a range variable to improve the readability of your queries. We will follow this convention in all our examples.

(*Q*16) *Find the sids of sailors who have TeseTved a red boat.*

```
SELECT    R.sid
FROM      Boats B, Reserves R
WHERE     B.bid = R.bid AND 8.color = 'red'
```

This query contains a join of two tables, followed by a selection on the color of boats. We can think of 13 and R as rows in the corresponding tables that

[3]The table name cannot be used as an implicit. range variable once a range variable is introduced for t.he relation.

'prove' that a sailor with sid R.sid reserved a reel boat B.bid. On our example instances *R2* and 83 (Figures 5.1 and 5.2), the answer consists of the *sids 22,* 31, and 64. If we want the names of sailors in the result, we must also consider the Sailors relation, since Reserves does not contain this information, as the next example illustrates.

(Q2) Find the names of sailors who have reserved a Ted boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats 13
WHERE   S.sid = R.sid AND R.bid = 13.bid AND B.color = 'red'
```

This query contains a join of three tables followed by a selection on the color of boats. The join with Sailors allows us to find the name of the sailor who, according to Reserves tuple R, has reserved a red boat described by tuple 13.

(Q3) Find the coloTS of boats reseTved by LubbeT.

```
SELECT  13.color
FROM    Sailors S, Reserves R, Boats 13
WHERE   S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber'
```

This query is very similar to the previous one. Note that in general there may be more than one sailor called Lubber (since *sname* is not a key for Sailors); this query is still correct in that it will return the colors of boats reserved by *some* Lubber, if there are several sailors called Lubber.

(Q4) Find the names of sailors who have Teserved at least one boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid
```

The join of Sailors and Reserves ensures that for each selected *sname*, the sailor has made some reservation. (If a sailor has not made a reservation, the second step in the conceptual evaluation strategy would eliminate all rows in the cross-product that involve this sailor.)

5.2.2 Expressions and Strings in the SELECT Command

SQL supports a more general version of the select-list than just a list of colu1nn8. Each item in a select-list can be of the form *expression* AS *column_name*, where *expression* is any arithmetic or string expression over column names (possibly prefixed by range variables) and constants, and *column_name* is a new name for this column in the output of the query. It can also contain *aggregates* such as *sum* and *count*, which we will discuss in Section 5.5. The SQL standard also includes expressions over date and time values, which we will not discuss. Although not part of the SQL standard, many implementations also support the use of built-in functions such as *sqrt, sin,* and *rnod.*

(Q17) Compute increments for the mtings of peTsons who have sailed two different boats on the same day.

```
SELECT  S.sname, S.rating+1 AS rating
FROM    Sailors S, Reserves R1, Reserves R2
WHERE   S.sid = R1.sid AND S.sid = R2.sid
        AND R1.day = R2.day AND R1.bid <> R2.bid
```

Also, each item in a *qualification* can be as general as *expTession1* = *expression2.*

```
SELECT  S1.sname AS name1, S2.sname AS name2
FROM    Sailors Sl, Sailors S2
WHERE   2*S1.rating = S2.rating-1
```

For string comparisons, we can use the comparison operators (=, <, >, etc.) with the ordering of strings determined alphabetically as usual. If we need to sort strings by an order other than alphabetical (e.g., sort strings denoting month names in the calendar order January, February, March, etc.), SQL supports a general concept of a collation, or sort order, for a character set. A collation allows the user to specify which characters are 'less than' which others and provides great flexibility in string manipulation.

In addition, SQL provides support for pattern matching through the LIKE operator, along with the use of the wild-card symbols % (which stands for zero or more arbitrary characters) and _ (which stands for exactly one, arbitrary, character). Thus, '_AB%' denotes a pattern matching every string that contains at least three characters, with the second and third characters being A and B respectively. Note that unlike the other comparison operators, blanks can be significant for the LIKE operator (depending on the collation for the underlying character set). Thus, *'Jeff'* = *'Jeff'* is true while *'Jeff'LIKE 'Jeff '* is false. An example of the use of LIKE in a query is given below.

(Q18) Find the ages of sailors wh08e name begins and ends with B and has at least three chamcters.

```
SELECT  S.age
```

Regular Expressions in SQL: Reflecting the increased importance of text data, SQL:1999 includes a more powerful version of the LIKE operator called SIMILAR. This operator allows a rich set of regular expressions to be used as patterns while searching text. The regular expressions are similar to those supported by the Unix operating system for string searches, although the syntax is a little different.

Relational Algebra and SQL: The set operations of SQL are available in relational algebra. The main difference, of course, is that they are *multiset* operations in SQL, since tables are multisets of tuples.

```
FROM    Sailors S
WHERE   S.sname LIKE 'B.%B'
```

The only such sailor is Bob, and his age is 63.5.

5.3 UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.[4] SQL also provides other set operations: IN (to check if an element is in a given set), op ANY, op ALL (to compare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section, and the other operations in Section 5.4.

Consider the following query:

(Q5) Find the names of sailors who have reserved a red or a green boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid
        AND (B.color = 'red' OR B.color = 'green')
```

[4]Note that although the SQL standard includes these operations, many systems currently support only UNION. Also. many systems recognize the keyword MINUS for EXCEPT.

This query is easily expressed using the OR connective in the WHERE clause. However, the following query, which is identical except for the use of 'and' rather than 'or' in the English version, turns out to be much more difficult:

(Q6) Find the names of sailors who have reserved both a red and a green boat.

If we were to just replace the use of OR in the previous query by AND, in analogy to the English statements of the two queries, we would retrieve the names of sailors who have reserved a boat that is both red and green. The integrity constraint that *bid* is a key for Boats tells us that the same boat cannot have two colors, and so the variant of the previous query with AND in place of OR will always return an empty answer set. A correct statement of Query Q6 using AND is the following:

```
SELECT  S.sname
FROM    Sailors S, Reserves RI, Boats BI, Reserves R2, Boats B2
WHERE   S.sid = Rl.sid AND R1.bid = Bl.bid
        AND S.sid = R2.sid AND R2.bid = B2.bid
        AND B1.color='red' AND B2.color = 'green'
```

We can think of RI and BI as rows that prove that sailor S.sid has reserved a red boat. R2 and B2 similarly prove that the same sailor has reserved a green boat. S.sname is not included in the result unless five such rows S, RI, BI, R2, and B2 are found.

The previous query is difficult to understand (and also quite inefficient to execute, as it turns out). In particular, the similarity to the previous OR query (Query Q5) is completely lost. A better solution for these two queries is to use UNION and INTERSECT.

The OR query (Query Q5) can be rewritten as follows:

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sicl = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT  S2.sname
FROM    Sailors S2, Boats B2, Reserves R2
WHERE   S2.sid = H2.sid AND R2.bid = B2.bicl AND B2.color = 'green'
```

This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats. In complete symmetry, the AND query (Query Q6) can be rewritten as follows:

```
SELECT  S.snarne
```

```
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT  S2.sname
FROM    Sailors S2, Boats B2, Reserves R2
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

This query actually contains a subtle bug-if there are two sailors such as Horatio in our example instances *B1, R2,* and 83, one of whom has reserved a red boat and the other has reserved a green boat, the name Horatio is returned even though no one individual called Horatio has reserved both a red and a green boat. Thus, the query actually computes sailor names such that some sailor with this name has reserved a red boat and some sailor with the same name (perhaps a different sailor) has reserved a green boat.

As we observed in Chapter 4, the problem arises because we are using *sname* to identify sailors, and *sname* is not a key for Sailors! If we select *sid* instead of *sname* in the previous query, we would compute the set of *sids* of sailors who have reserved both red and green boats. (To compute the names of such sailors requires a nested query; we will return to this example in Section 5.4.4.)

Our next query illustrates the set-difference operation in SQL.

(*Q*19) *Find the* sids *of all sailor's who have reserved red boats but not green boats.*

```
SELECT  S.sid
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT  S2.sid
FROM    Sailors S2, Reserves R2, Boats B2
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Sailors 22, 64, and 31 have reserved red boats. Sailors 22, 74, and 31 have reserved green boats. Hence, the answer contains just the *sid 64.*

Indeed, since the Reserves relation contains sid information, there is no need to look at the Sailors relation, and we can use the following simpler query:

```
SELECT  R.sid
FROM    Boats B, Reserves R
WHERE   R.bicl = B.bid AND B.color = 'red'
EXCEPT
```

```
SELECT R2.sid
FROM   Boats B2, Reserves R2
WHERE  R2.bicl = B2.bid AND B2.color = :green'
```

Observe that this query relies on referential integrity; that is, there are no reservations for nonexisting sailors. Note that UNION, INTERSECT, and EXCEPT can be used on *any* two tables that are union-compatible, that is, have the same number of columns and the columns, taken in order, have the same types. For example, we can write the following query:

(Q20) Find all sids of sailors who have a rating of 10 or *reserved boat 104.*

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating = 10
UNION
SELECT R.sid
FROM   Reserves R
WHERE  R.bid = 104
```

The first part of the union returns the *sids* 58 and 71. The second part returns 22 and 31. The answer is, therefore, the set of *sids* 22, 31, 58, and 71. A final point to note about UNION, INTERSECT, and EXCEPT follows. In contrast to the default that duplicates are not eliminated unless DISTINCT is specified in the basic query form, the default for UNION queries is that duplicates *are* eliminated! To retain duplicates, UNION ALL must be used; if so, the number of copies of a row in the result is always $m + n$, where m and n are the numbers of times that the row appears in the two parts of the union. Similarly, INTERSECT ALL retains cluplicates--the number of copies of a row in the result is $min(m, n)$—and EXCEPT ALL also retains duplicates—the number of copies of a row in the result is $m - n$, where 'm corresponds to the first relation.

5.4 NESTED QUERIES

One of the most powerful features of SQL is nested queries. A **nested query** is a query that has another query embedded within it; the embedded query is called a **suhquery**. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. When writing a query, we sornetimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query. A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause (which we present in Section 5.5).

Relational Algebra and SQL: Nesting of queries is a feature that is not available in relational algebra, but nested queries can be translated into algebra, as we will see in Chapter 15. Nesting in SQL is inspired more by relational calculus than algebra. In conjunction with some of SQL's other features, such as (multi)set operators and aggregation, nesting is a very expressive construct.

This section discusses only subqueries that appear in the WHERE clause. The treatment of subqueries appearing elsewhere is quite similar. Some examples of subqueries that appear in the FROM clause are discussed later in Section 5.5.1.

5.4.1 Introduction to Nested Queries

As an example, let us rewrite the following query, which we discussed earlier, using a nested subquery:

(Ql) Find the names of sailors who have reserved boat 103.

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN ( SELECT  R.sid
                   FROM    Reserves R
                   WHERE   R.bid = 103 )
```

The nested subquery computes the (multi)set of *sids* for sailors who have reserved boat 103 (the set contains 22,31, and 74 on instances *R2* and 83), and the top-level query retrieves the names of sailors whose *sid* is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested. Note that it is very easy to modify this query to find all sailors who have *not* reserved boat 103-we can just replace IN by NOT IN!

The best way to understand a nested query is to think of it in terms of a conceptual evaluation strategy. In our example, the strategy consists of examining rows in Sailors and, for each such row, evaluating the subquery over Reserves. In general, the conceptual evaluation strategy that we presented for defining the semantics of a query can be extended to cover nested queries as follows: Construct the cross-product of the tables in the FROM clause of the top-level query as hefore. For each row in the cross-product, while testing the qllalification in the WHERE clause, (re)compute the subquery.5 Of course, the subquery might itself contain another nested subquery, in which case we apply the same idea one more time, leading to an evaluation strategy with several levels of nested loops.

As an example of a multiply nested query, let us rewrite the following query.

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM   Reserves R
                  WHERE  R.bid IN (SELECT B.bid
                                   FROM   Boats B
                                   WHERE  B.color = 'red'
```

The innermost subquery finds the set of *bids* of red boats (102 and 104 on instance *E1)*. The subquery one level above finds the set of *sids* of sailors who have reserved one of these boats. On instances *E1, R2,* and 83, this set of *sids* contains 22, 31, and 64. The top-level query finds the names of sailors whose *sid* is in this set of *sids*; we get Dustin, Lubber, and Horatio.

To find the names of sailors who have not reserved a red boat, wc replace the outermost occurrence of IN by NOT IN, as illustrated in the next query.

(Q21) Find the names of sailors who have not *reserved a red boat.*

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid NOT IN ( SELECT R.sid
                      FROM   Reserves R
                      WHERE  R.bid IN ( SELECT B.bid
                                        FROM   Boats B
                                        WHERE  B.color = 'red' )
```

This qucry computes the names of sailors whose *sid* is *not* in the set 22, 31, and 64.

In contrast to Query Q21, we can modify the previous query (the nested version of Q2) by replacing the inner occurrence (rather than the outer occurence) of

5Since the inner subquery in our example does not depend on the 'current' row from the outer query ill any way, you rnight wonder why we have to recompute the subquery for each outer row. For an answer, see Section 5.4.2.

IN with NOT IN. This modified query would compute the names of sailors who have reserved a boat that is not red, that is, if they have a reservation, it is not for a red boat. Let us consider how. In the inner query, we check that *R.bid* is *not* either 102 or 104 (the *bids* of red boats). The outer query then finds the *sids* in Reserves tuples where the *bid* is not 102 or 104. On instances *B*1, *R*2, and 53, the outer query computes the set of *sids* 22, 31, 64, and 74. Finally, we find the names of sailors whose *sid* is in this set.

\Ve can also modify the nested query Q2 by replacing both occurrences of IN with NOT IN. This variant finds the names of sailors who have not reserved a boat that is not red, that is, who have reserved only red boats (if they've reserved any boats at all). Proceeding as in the previous paragraph, on instances *E1, R2,* and 53, the outer query computes the set of *sids* (in Sailors) other than 22, 31, 64, and 74. This is the set 29, 32, 58, 71, 85, and 95. We then find the names of sailors whose *sid* is in this set.

5.4.2 Correlated Nested Queries

In the nested queries seen thus far, the inner subquery has been completely independent of the outer query. In general, the inner subquery could depend on the row currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more.

(*Q*1) *Find the names of sailors who have reserved boat number 103.*

```
SELECT S.sname
FROM   Sailors S
WHERE  EXISTS ( SELECT *
                FROM   Reserves R
                WHERE  R.bid = 103
                       AND R.sid = S.sid )
```

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set. Thus, for each Sailor row 5, we test whether the set of Reserves rows *R* such that *R.bid = 103* AND *S.sid = R.sid* is nonempty. If so, sailor 5 has reserved boat 103, and we retrieve the name. '1'he subquery clearly depends on the current row Sand IIlUSt be re-evaluated for each row in Sailors. The occurrence of *S* in the subquery (in the form of the literal *S.sid)* is called a *cOTTelation,* and such queries are called *correlated queries.*

This query also illustrates the use of the special symbol * in situations where all we want to do is to check that a qualifying row exists, and do Hot really want to retrieve any columns from the row. This is one of the two uses of * in the SELECT clause that is good programming style; the other is as an argument of the COUNT aggregate operation, which we describe shortly.

As a further example, by using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat. Closely related to EXISTS is the UNIQUE predicate. \Vhen we apply UNIQUE to a subquery, the resulting condition returns true if no row appears twice in the answer to the subquery, that is, there are no duplicates; in particular, it returns true if the answer is empty. (And there is also a NOT UNIQUE version.)

5.4.3 Set-Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. (SOME is also available, but it is just a synonym for ANY.)

(Q22) Find sailors whose rating is better than some sailor called Horatio.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating > ANY ( SELECT  S2.rating
                         FROM    Sailors S2
                         WHERE   S2.sname = 'Horatio' )
```

If there are several sailors called Horatio, this query finds all sailors whose rating is better than that of *some* sailor called Horatio. On instance 83, this computes the *sids* 31, 32, 58, 71, and 74. What if there were *no* sailor called Horatio? In this case the comparison *S.rating* > ANY ... is defined to return false, and the query returns an elnpty answer set. To understand comparisons involving ANY, it is useful to think of the comparison being carried out repeatedly. In this example, *S. rating* is successively compared with each rating value that is an answer to the nested query. Intuitively, the subquery must return a row that makes the comparison true, in order for S. *rat'ing* > ANY ... to return true.

(Q23) *Find sailors whose rating is better than every sailor' called Horat·to.*

We can obtain all such queries with a simple modification to Query Q22: Just replace ANY with ALL in the WHERE clause of the outer query. On instance $S3$, we would get the *sids* 58 and 71. If there were no sailor called Horatio, the comparison *S.rating* > ALL ... is defined to return true! The query would then return the names of all sailors. Again, it is useful to think of the comparison being carried out repeatedly. Intuitively, the comparison must be true for every returned row for *S.rating*> ALL ... to return true.

As another illustration of ALL, consider the following query.

(Q24J Find the sailors with the highest rating.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating >= ALL ( SELECT  S2.rating
                          FROM    Sailors S2 )
```

The subquery computes the set of all rating values in Sailors. The outer WHERE condition is satisfied only when *S.rating* is greater than or equal to each of these rating values, that is, when it is the largest rating value. In the instance 53, the condition is satisfied only for *rating* 10, and the answer includes the *sids* of sailors with this rating, Le., 58 and 71.

Note that IN and NOT IN are equivalent to = ANY and <> ALL, respectively.

5.4.4 More Examples of Nested Queries

Let us revisit a query that we considered earlier using the INTERSECT operator.

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
        AND S.sid IN ( SELECT  S2.sid
                       FROM    Sailors S2, Boats B2, Reserves R2
                       WHERE   S2.sid = R2.sid AND R2.bid = B2.bid
                               AND B2.color = 'green' )
```

This query can be understood as follows: "Find all sailors who have reserved a red boat and, further, have *sids* that are included in the set of *sids* of sailors who have reserved a green boat." This formulation of the query illustrates how queries involving INTERSECT can be rewritten using IN, which is useful to know if your system does not support INTERSECT. Queries using EXCEPT can be similarly rewritten by using NOT IN. To find the *sids* of sailors who have reserved red boats but not green boats, we can simply replace the keyword IN in the previous query by NOT IN.

As it turns out, writing this query (Q6) using INTERSECT is more complicated because we have to use *sids* to identify sailors (while intersecting) and have to return sailor names:

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN (( SELECT R.sid
                   FROM   Boats B, Reserves R
                   WHERE  R.bid = B.bid AND B.color = 'red' )
                   INTERSECT
                   (SELECT R2.sid
                   FROM   Boats B2, Reserves R2
                   WHERE  R2.bid = B2.bid AND B2.color = 'green' ))
```

Our next example illustrates how the *division* operation in relational algebra can be expressed in SQL.

(Q9) Find the names of sailors who have TeseTved all boats.

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS (( SELECT B.bid
                     FROM   Boats B )
                     EXCEPT
                     (SELECT R.bid
                     FROM   Reserves R
                     WHERE  R.sid = S.sid ))
```

Note that this query is correlated--for each sailor *S*, we check to see that the set of boats reserved by *S* includes every boat. An alternative way to do this query without using EXCEPT follows:

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS ( SELECT B.bid
                    FROM   Boats B
                    WHERE  NOT EXISTS ( SELECT R.bid
                                        FROM   Reserves R
                                        WHERE  R.bid = B.bid
                                               AND R.sid = S.sid ))
```

Intuitively, for each sailor we check that there is no boat that has not been reserved by this sailor.

SQL:1999 Aggregate **Functions:** The collection of aggregate functions is greatly expanded in the new standard, including several statistical functions such as standard deviation, covariance, and percentiles. However, the new aggregate functions are in the SQLjOLAP package and may not be supported by all vendors.

5.5 AGGREGATE OPERATORS

In addition to simply retrieving data, we often want to perform some computation or summarization. As we noted earlier in this chapter, SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Note that it does not make sense to specify DISTINCT in conjunction with MIN or MAX (although SQL does not preclude this).

(Q25) Find the average age of all sailors.

```
SELECT AVG (S.age)
FROM   Sailors S
```

On instance 53, the average age is 37.4. Of course, the WHERE clause can be used to restrict the sailors considered in computing the average age.

(Q26) Find the average age of sailors with a rating of 10.

```
SELECT AVG (S.age)
FROM   Sailors S
WHERE  S.rating = 10
```

There are two such sailors, and their average age is 25.5. MIN (or MAX) can be used instead of AVG in the above queries to find the age of the youngest (oldest) sailor. However) finding both the name and the age of the oldest sailor is more tricky, as the next query illustrates.

(Q27) Find the name and age of the oldest sailor.

Consider the following attempt to answer this query:

```
SELECT S.sname, MAX (S.age)
FROM   Sailors S
```

The intent is for this query to return not only the maximum age but also the name of the sailors having that age. However, this query is illegal in SQL-if the SELECT clause uses an aggregate operation, then it must use *only* aggregate operations unless the query contains a GROUP BY clause! (The intuition behind this restriction should become clear when we discuss the GROUP BY clause in Section 5.5.1.) Therefore, we cannot use MAX (S.age) as well as S.sname in the SELECT clause. We have to use a nested query to compute the desired answer to Q27:

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.age = ( SELECT MAX (S2.age)
                 FROM Sailors S2 )
```

Observe that we have used the result of an aggregate operation in the subquery as an argument to a comparison operation. Strictly speaking, we are comparing an age value with the result of the subquery, which is a relation. However, because of the use of the aggregate operation, the subquery is guaranteed to return a single tuple with a single field, and SQL converts such a relation to a field value for the sake of the comparison. The following equivalent query for Q27 is legal in the SQL standard but, unfortunately, is not supported in many systems:

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  ( SELECT MAX (S2.age)
         FROM Sailors S2 ) = S.age
```

We can count the number of sailors using COUNT. This example illustrates the use of * as an argument to COUNT, which is useful when \ve want to count all rows.

(Q28) Count the number of sailors.

```
SELECT COUNT (*)
```

```
FROM    Sailors S
```

We can think of * as shorthand for all the columns (in the cross-product of the **from-list** in the FROM clause). Contrast this query with the following query, which computes the number of distinct sailor names. (Remember that *sname* is not a key!)

(Q29) Count the number of different sailor names.

```
SELECT  COUNT ( DISTINCT S.sname )
FROM    Sailors S
```

On instance 83, the answer to Q28 is 10, whereas the answer to Q29 is 9 (because two sailors have the same name, Horatio). If DISTINCT is omitted, the answer to Q29 is 10, because the name Horatio is counted twice. If COUNT does not include DISTINCT, then COUNT (*) gives the same answer as COUNT *(x)* , where *x* is any set of attributes. In our example, without DISTINCT Q29 is equivalent to Q28. However, the use of COUNT (*) is better querying style, since it is immediately clear that all records contribute to the total count.

Aggregate operations offer an alternative to the ANY and ALL constructs. For example, consider the following query:

(Q30) Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age > ( SELECT  MAX ( S2.age )
                  FROM    Sailors S2
                  WHERE   S2.rating = 10 )
```

On instance 83, the oldest sailor with rating 10 is sailor 58, whose age is 35. The names of older sailors are Bob, Dustin, Horatio, and Lubber. Using ALL, this query could alternatively be written as follows:

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age > ALL ( SELECT  S2.age
                      FROM    Sailors S2
                      WHERE   S2.rating = 10 )
```

However, the ALL query is more error proncone could easily (and incorrectly!) use ANY instead of ALL, and retrieve sailors who are older than *some* sailor with

Relational Algebra and SQL: Aggregation is a fundamental operation that canIlot be expressed in relational algebra. Similarly, SQL's grouping construct cannot be expressed in algebra.

a rating of 10. The use of ANY intuitively corresponds to the use of MIN, instead of MAX, in the previous query.

5.5.1 The **GROUP BY and HAVING Clauses**

Thus far, we have applied aggregate operations to all (qualifying) rows in a relation. Often we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance). For example, consider the following query.

(Q31) Find the age of the youngest sailor for each rating level.

If we know that ratings are integers in the range 1 to la, we could write 10 queries of the form:

```
SELECT  MIN (S.age)
FROM    Sailors S
WHERE   S.rating = i
```

where i = 1,2, ... ,10. Writing 10 such queries is tedious. More important, we may not know what rating levels exist in advance.

To write such queries, we need a major extension to the basic SQL query form, namely, the GROUP BY clause. In fact, the extension also includes an optional HAVING clause that can be used to specify qualificatioIls over groups (for example, we may be interested only in rating levels> 6. The general form of an SQL query with these extensions is:

```
SELECT     [ DISTINCT] select-list
FROM       from-list
WHERE     'qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

Using the GROUP BY clause, we can write Q31 as follows:

```
SELECT     S.rating, MIN (S.age)
```

```
FROM      Sailors S
GROUP BY  S.rating
```

Let us consider some important points concerning the new clauses:

- The select-list in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form aggop (*column-name*) AS *new-name*. We already saw AS used to rename output columns. Columns that are the result of aggregate operators do not already have a column name, and therefore giving the column a name with AS is especially useful.

  Every column that appears in (1) must also appear in grouping-list. The reason is that each row in the result of the query corresponds to one *group*, which is a collection of rows that agree on the values of columns in grouping-list. In general, if a column appears in list (1), but not in grouping-list, there can be multiple rows within a group that have different values in this column, and it is not clear what value should be assigned to this column in an answer row.

  We can sometimes use primary key information to verify that a column has a unique value in all rows within each group. For example, if the grouping-list contains the primary key of a table in the from-list, every column of that table has a unique value within each group. In SQL:1999, such columns are also allowed to appear in part (1) of the select-list.

- The expressions appearing in the group-qualification in the HAVING clause must have a *single* value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. To satisfy this requirement in SQL-92, a column appearing in the group-qualification must appear as the argument to an aggregation operator, or it must also appear in grouping-list. In SQL:1999, two new set functions have been introduced that allow us to check whether *every* or *any* row in a group satisfies a condition; this allows us to use conditions similar to those in a WHERE clause.

- If GROUP BY is omitted, the entire table is regarded as a single group.

We explain the semantics of such a query through an example.

(Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

```
SELECT    S.rating, MIN (S.age) AS minage
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

We will evaluate this query on instance 83 of Sailors, reproduced in Figure 5.10 for convenience. The instance of Sailors on which this query is to be evaluated is shown in Figure 5.10. Extending the conceptual evaluation strategy presented in Section 5.2, we proceed as follows. The first step is to construct the cross-product of tables in the from-list. Because the only relation in the from-list in Query Q32 is Sailors, the result is just the instance shown in Figure 5.10.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5
96	Frodo	3	25.5

Figure 5.10 Instance 53 of Sailors

The second step is to apply the qualification in the WHERE clause, *S.age* >= 18. This step eliminates the row (71, *zorba*, 10, 16). The third step is to eliminate unwanted columns. Only columns mentioned in the SELECT clause, the GROUP BY clause, or the HAVING clause are necessary, which means we can eliminate *sid* and *sname* in our example. The result is shown in Figure 5.11. Observe that there are two identical rows with *rating* 3 and *age* 25.5-SQL does not eliminate duplicates except when required to do so by use of the DISTINCT keyword! The number of copies of a row in the intermediate table of Figure 5.11 is determined by the number of rows in the original table that had these values in the projected columns.

The fourth step is to sort the table according to the GROUP BY clause to identify the groups. The result of this step is shown in Figure 5.12.

The fifth step is to apply the group-qualification in the HAVING clause, that is, the condition COUNT (*) > 1. This step eliminates the groups with *rating* equal to 1, 9, and 10. Observe that the order in which the WHERE and GROUP BY clauses are considered is significant: If the WHERE clause were not considered first, the group with *rating=10* would have met the group-qualification in the HAVING clause. The sixth step is to generate one answer row for each remaining group. The answer row corresponding to a group consists of a subset

rating	*age*
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
9	35.0
3	25.5
3	63.5
3	25.5

Figure 5.11 After Evaluation Step 3

rating	*age*
1	33.0
3	25.5
3	25.5
3	63.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

Figure 5.12 After Evaluation Step 4

of the grouping columns, plus one or more columns generated by applying an aggregation operator. In our example, each answer row has a *rating* column and a *minage* column, which is computed by applying MIN to the values in the *age* column of the corresponding group. The result of this step is shown in Figure 5.13.

rating	*minage*
3	25.5
7	35.0
8	25.5

Figure 5.13 Final Result in Sample Evaluation

If the query contains DISTINCT in the SELECT clause, duplicates are eliminated in an additional, and final, step.

SQL:1999 has introduced two new set functions, EVERY and ANY. To illustrate these functions, we can replace the HAVING clause in our example by

```
HAVING   COUNT (*) > 1 AND EVERY ( S.age <= 60 )
```

The fifth step of the conceptual evaluation is the one affected by the change in the HAVING clause. Consider the result of the fourth step, shown in Figure 5.12. The EVERY keyword requires that every row in a group must satisfy the attached condition to meet the group-qualification. The group for *rating* 3 does meet this criterion and is dropped; the result is shown in Figure 5.14.

SQL:1999 Extensions: Two new set functions, EVERY and ANY, have been added. When they are used in the HAVING clause, the basic intuition that the clause specifies a condition to be satisfied by each group, taken as a whole, remains unchanged. However, the condition can now involve tests on individual tuples in the group, whereas it previously relied exclusively on aggregate functions over the group of tuples.

It is worth contrasting the preceding query with the following query, in which the condition on *age* is in the WHERE clause instead of the HAVING clause:

```
SELECT    S.rating, MIN (S.age) AS minage
FROM      Sailors S
WHERE     S.age >= 18 AND S.age <= 60
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

Now, the result after the third step of conceptual evaluation no longer contains the row with *age* 63.5. Nonetheless, the group for *rating* 3 satisfies the condition COUNT (*) > 1, since it still has two rows, and meets the group-qualification applied in the fifth step. The final result for this query is shown in Figure 5.15.

rating	*minage*
7	45.0
8	55.5

Figure 5.14 Final Result of EVERY Query

rating	*minage*
3	25.5
7	45.0
8	55.5

Figure 5.15 Result of Alternative Query

5.5.2 More Examples of Aggregate Queries

(Q33) For each red boat; find the number of reservations for this boat.

```
SELECT    B.bid, COUNT (*) AS reservationcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid AND B.color = 'red'
GROUP BY  B.bid
```

On instances *B1* and *R2,* the answer to this query contains the two tuples (102, 3) and (104, 2).

Observe that this version of the preceding query is illegal:

```
SELECT    B.bicl, COUNT (*) AS reservationcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid
GROUP BY  B.bid
HAVING    B.color = 'red'
```

Even though the gToup-qualification *B.coloT = 'Ted'is* single-valued per group, since the grouping attribute *bid* is a key for Boats (and therefore determines *coloT)*, SQL disallows this query.6 Only columns that appear in the GROUP BY clause can appear in the HAVING clause, unless they appear as arguments to an aggregate operator in the HAVING clause.

(Q34) Find the avemge age of sailoTs fOT each rating level that has at least two sailoTs.

```
SELECT    S.rating, AVG (S.age) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

After identifying groups based on *mting*, we retain only groups with at least two sailors. The answer to this query on instance 83 is shown in Figure 5.16.

mting	*avgage*
3	44.5
7	40.0
8	40.5
10	25.5

Figure 5.16 Q34 Answer

mting	*avgage*
3	45.5
7	40.0
8	40.5
10	35.0

Figure 5.17 Q35 Answer

rating	*avgage*
3	45.5
7	40.0
8	40.5

Figure 5.18 Q36 Answer

The following alternative formulation of Query Q34 illustrates that the HAVING clause can have a nested subquery, just like the WHERE clause. Note that we can use *S.rating* inside the nested subquery in the HAVING clause because it has a single value for the current group of sailors:

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

6This query can be easily rewritten to be legal in SQL:1999 using EVERY in the HAVING clause.

(Q35) Find the average age of sailors who aTe of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age >= 18
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

In this variant of Query Q34, we first remove tuples with *age* <= 18 and group the remaining tuples by *rating*. For each group, the subquery in the HAVING clause computes the number of tuples in Sailors (without applying the selection *age* <= 18) with the same *rating* value as the current group. If a group has less than two sailors, it is discarded. For each remaining group, we output the average age. The answer to this query on instance 53 is shown in Figure 5.17. Note that the answer is very similar to the answer for Q34, with the only difference being that for the group with rating 10, we now ignore the sailor with age 16 while computing the average.

(Q36) Find the average age oj sailors who aTe of voting age (i.e., at least 18 yeaTs old) JOT each rating level that has at least two such *sailors.*

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age> 18
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating AND S2.age >= 18 )
```

This formulation of the query reflects its similarity to Q35. The answer to Q36 on instance 53 is shown in Figure 5.18. It differs from the answer to Q35 in that there is no tuple for rating 10, since there is only one tuple with rating 10 and *age* $\geq$ 18.

Query Q36 is actually very similar to Q32, as the following simpler formulation shows:

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age> 18
GROUP BY  S.rating
```

```
HAVING   COUNT (*) > 1
```

This formulation of Q36 takes advantage of the fact that the WHERE clause is applied before grouping is done; thus, only sailors with *age*> 18 are left when grouping is done. It is instructive to consider yet another way of writing this query:

```
SELECT Temp.rating, Temp.avgage
FROM   ( SELECT     S.rating, AVG ( S.age ) AS avgage,
                    COUNT (*) AS ratingcount
         FROM       Sailors S
         WHERE      S. age> 18
         GROUP BY   S.rating) AS Temp
WHERE Temp.ratingcount > 1
```

This alternative brings out several interesting points. First, the FROM clause can also contain a nested subquery according to the SQL standard.[7] Second, the HAVING clause is not needed at all. Any query with a HAVING clause can be rewritten without one, but many queries are simpler to express with the HAVING clause. Finally, when a subquery appears in the FROM clause, using the AS keyword to give it a name is necessary (since otherwise we could not express, for instance, the condition *Temp. ratingcount* > 1).

(Q37) Find those ratings for which the average age of sailors is the m'inirnum over all ratings.

We use this query to illustrate that aggregate operations cannot be nested. One might consider writing it as follows:

```
SELECT   S.rating
FROM     Sailors S
WHERE    AVG (S.age) = ( SELECT    MIN (AVG (S2.age))
                         FROM      Sailors S2
                         GROUP BY  S2.rating )
```

A little thought shows that this query will not work even if the expression MIN (AVG (S2.age)), which is illegal, were allowed. In the nested query, Sailors is partitioned into groups by rating, and the average age is computed for each rating value. For each group, applying MIN to this average age value for the group will return the same value! A correct version of this query follows. It essentially computes a temporary table containing the average age for each rating value and then finds the rating(s) for which this average age is the minimum.

7Not all commercial database systems currently support nested queries in the FROM clause.

The Relational Model and SQL: Null values are not part of the basic relational model. Like SQL's treatment of tables as multisets of tuples, this is a departure from the basic model.

```
SELECT  Temp.rating, Temp.avgage
FROM    ( SELECT    S.rating, AVG (S.age) AS avgage,
          FROM      Sailors S
          GROUP BY  S.rating) AS Temp
WHERE   Temp.avgage = ( SELECT MIN (Temp.avgage) FROM Temp)
```

The answer to this query on instance 53 is ⟨10, 25.5⟩.

As an exercise, consider whether the following query computes the same answer.

```
SELECT    Temp.rating, MIN (Temp.avgage )
FROM      ( SELECT    S.rating, AVG (S.age) AS avgage,
            FROM      Sailors S
            GROUP BY  S.rating) AS Temp
GROUP BY  Temp.rating
```

5.6 NULL VALUES

Thus far, we have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a *rating* column, what row should we insert for Dan? What is needed here is a special value that denotes *unknown.* Suppose the Sailor table definition was modified to include a *rnaiden-name* column. However, only married women who take their husband's last name have a maiden name. For women who do not take their husband's name and for men, the *maiden-name* column is *inapplicable.* Again, what value do we include in this column for the row representing Dan?

SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable.* Using our Sailor table definition, we might enter the row (98. *Dan, null*, 39) to represent Dan. The presence of *null* values complicates rnany issues, and we consider the impact of *null* values on SQL in this section.

5.6.1 Comparisons Using Null Values

Consider a comparison such as *rating* = 8. If this is applied to the row for Dan, is this condition true or false? Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the case for the comparisons *rating*> 8 and *rating* < 8 as well. Perhaps less obviously, if we compare two *null* values using <, >, =, and so on, the result is always unknown. For example, if we have *null* in two distinct rows of the sailor relation, any comparison returns unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null*; for example, we can say *rating* IS NULL, which would evaluate to true on the row representing Dan. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for Dan.

5.6.2 Logical Connectives AND, OR, and NOT

Now, what about boolean expressions such as *rating* = 8 OR *age* < 40 and *mting* = 8 AND *age* < 40? Considering the row for Dan again, because *age* < 40, the first expression evaluates to true regardless of the value of *rating*, but what about the second? We can only say unknown.

But this example raises an important point—once we have *null* values, we must define the logical operators AND, OR, and NOT using a *three-valued* logic in which expressions evaluate to true, false, or unknown. We extend the usu1'11 interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown as follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. (If both arguments are false, of course, OR evaluates to false.) AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown. (If both arguments are true, AND evaluates to true.)

5.6.3 Impact on SQL Constructs

Boolean expressions arise in many contexts in SQL, and the impact of *null* values must be recognized. For example, the qualification in the WHERE clause eliminates rows (in the cross-product of tables named in the FROM clause) for which the qualification does not evaluate to true. Therefore, in the presence of *null* values, any row that evaluates to false or unknown is eliminated. Eliminating rows that evaluate to unknown has a subtle but signifieant impaet on queries, especially nested queries involving EXISTS or UNIQUE.

Another issue in the presence of *null* values is the definition of when two rows in a relation instance are regarded as *duplicates*. The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain *null*. Contrast this definition with the fact that if we compare two *null* values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly.

As expected, the arithmetic operations +, -, *, and / all return *null* if one of their arguments is *null*. However, nulls can cause some unexpected behavior with aggregate operations. COUNT(*) handles *null* values just like other values; that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard *null* values—thus SUM cannot be understood as just the addition of all values in the (multi)set of values that it is applied to; a preliminary step of discarding all *null* values must also be accounted for. As a special case, if one of these operators-other than COUNT-is applied to *only* null values, the result is again *null*.

5.6.4 Outer Joins

Some interesting variants of the join operation that rely on *null* values, called outer joins, are supported in SQL. Consider the join of two tables, say Sailors $\bowtie_c$ Reserves. Tuples of Sailors that do not match some row in Reserves according to the join condition c do not appear in the result. In an outer join, on the other hanel, Sailor rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned *null* values.

In fact, there are several variants of the outer join idea. In a left outer join, Sailor rows without a matching Reserves row appear in the result, but not vice versa. In a right outer join, Reserves rows without a matching Sailors row appear in the result, but not vice versa. In a **full** outer join, both Sailors and Reserves rows without a match appear in the result. (Of course, rows with a match always appear in the result, for all these variants, just like the usual joins, sometimes called *inner* joins, presented in Chapter 4.)

SQL allows the desired type of join to be specified in the FROM clause. For example, the following query lists *(sid, bid)* pairs corresponding to sailors and boats they have reserved:

```
SELECT S.sid, R.bid
FROM   Sailors S NATURAL LEFT OUTER JOIN Reserves R
```

The NATURAL keyword specifies that the join condition is equality on all common attributes (in this example, *sid*), and the WHERE clause is not required (unless we want to specify additional, non-join conditions). On the instances of Sailors and Reserves shown in Figure 5.6, this query computes the result shown in Figure 5.19.

sid	*bid*
22	101
31	*null*
58	103

Figure 5.19 Left Outer Join of *Sailor1* and *Reserves1*

5.6.5 Disallowing Null Values

We can disallow *null* values by specifying NOT NULL as part of the field definition; for example, *sname* CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on *null* values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

Our coverage of *null* values is far from complete. The interested reader should consult one of the many books devoted to SQL for a more detailed treatment of the topic.

5.7 COMPLEX INTEGRITY CONSTRAINTS IN SQL

In this section we discuss the specification of complex integrity constraints that utilize the full power of SQL queries. The features discussed in this section complement the integrity constraint features of SQL presented in Chapter 3.

5.7.1 Constraints over a Single Table

We can specify complex constraints over a single table using **table constraints**, which have the form CHECK *conditional-expression*. For example, to ensure that *rating* must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                       rating  INTEGER,
                       age     REAL,
                       PRIMARY KEY (sid),
                       CHECK (rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

```
CREATE TABLE Reserves (sid   INTEGER,
                       bid   INTEGER,
                       day   DATE,
                       FOREIGN KEY (sid) REFERENCES Sailors
                       FOREIGN KEY (bid) REFERENCES Boats
                       CONSTRAINT noInterlakeRes
                       CHECK ( 'Interlake' <>
                               ( SELECT B.bname
                                 FROM   Boats B
                                 WHERE  B.bid = Reserves.bid )))
```

When a row is inserted into Reserves or an existing row is modified, the *conditional expression* in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

5.7.2 Domain Constraints and Distinct Types

A user can define a new domain using the CREATE DOMAIN statement, which uses CHECK constraints.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
                        CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the underlying, or *source*, type for the domain ratingval, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint; in defining this constraint, we use the keyword VALUE to refer to a value in the domain. By using this facility, we can constrain the values that belong to a domain using the full power of SQL queries. Once a domain is defined, the name of the domain can be used to restrict column values in a table; we can use the following line in a schema declaration, for example:

```
rating  ratingval
```

The optional DEFAULT keyword is used to associate a default value with a domain. If the domain ratingval is used for a column in some relation and no value is entered for this column in an inserted tuple, the default value 1 associated with ratingval is used.

SQL's support for the concept of a domain is limited in an important respect. For example, we can define two domains called SailorId and BoatId, each

SQL:1999 Distinct Types: :Many systems, e.g., Informix UDS and IBM DB2, already support this feature. With its introduction, we expect that the support for domains will be *deprecated*, and eventually eliminated, in future versions of the SQL standard. It is really just one part of a broad set of object-oriented features in SQL:1999, which we discuss in Chapter 23.

using INTEGER as the underlying type. The intent is to force a comparison of a SailorId value with a BoatId value to always fail (since they are drawn from different domains); however, since they both have the same base type, INTEGER, the comparison will succeed in SQL. This problem is addressed through the introduction of distinct types in SqL:1999:

```
CREATE TYPE ratingtype AS INTEGER
```

This statement defines a new *distinct* type called ratingtype, with INTEGER as its source type. Values of type ratingtype can be compared with each other, but they cannot be compared with values of other types. In particular, ratingtype values are treated as being distinct from values of the source type, INTEGER—we cannot compare them to integers or combine them with integers (e.g., add an integer to a ratingtype value). If we want to define operations on the new type, for example, an *average* function, we must do so explicitly; none of the existing operations on the source type carryover. We discuss how such functions can be defined in Section 23.4.1.

5.7.3 Assertions: ICs over Several Tables

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold *only* if the a,ssociated table is nonempty. Thus, when a constraint involves two or more tables, the table constraint mechanism is sometimes cumbersome and not quite what is desired. To cover such situations, SQL supports the creation of assertions, which are constraints not associated with anyone table.

As an example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100. (This condition Illight be required, say, to qualify as a 'small' sailing club.) We could try the following table constraint:

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                    rating  INTEGER,
                    age     REAL,
                    PRIMARY KEY (sid),
                    CHECK ( rating >= 1 AND rating <= 10)
                    CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S )
                            + ( SELECT COUNT (B.bid) FROM Boats B )
                            < 100 ))
```

This solution suffers from two drawbacks. It is associated with Sailors, although it involves Boats in a completely symmetric way. More important, if the Sailors table is empty, this constraint is defined (as per the semantics of table constraints) to always hold, even if we have more than 100 rows in Boats! We could extend this constraint specification to check that Sailors is nonempty, but this approach becomes cumbersome. The best solution is to create an assertion, as follows:

```
CREATE ASSERTION smallClub
CHECK (( SELECT COUNT (S.sid) FROM Sailors S )
        + ( SELECT COUNT (B.bid) FROM Boats B)
        < 100 )
```

5.8 TRIGGERS AND ACTIVE DATABASES

A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an **active database**. A trigger description contains three parts:

- **Event:** A change to the database that activates the trigger.
- **Condition:** A query or test that is run when the trigger is activated.
- **Action:** A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the *event* specification. An insert, delete, or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.

A *condition* in a trigger can be a true/false statement (e.g., all employee salaries are less than $100,000) or a query. A query is interpreted as *true* if the answer set is nonempty and *false* if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.

A trigger *action* can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute Hew queries, and make changes to the database. In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit) or call host-language procedures.

An important issue is when the action part of a trigger executes in relation to the statement that activated the trigger. For example, a statement that inserts records into the Students table may activate a trigger that is used to maintain statistics on how many students younger than 18 are inserted at a time by a typical insert statement. Depending on exactly what the trigger does, we may want its action to execute *before* changes are made to the Students table or *afterwards:* A trigger that initializes a variable used to count the nurnber of qualifying insertions should be executed before, and a trigger that executes once per qualifying inserted record and increments the variable should be executed after each record is inserted (because we may want to examine the values in the new record to determine the action).

5.8.1 Examples of Triggers in SQL

The examples shown in Figure 5.20, written using Oracle Server syntax for defining triggers, illustrate the basic concepts behind triggers. (The SQL:1999 syntax for these triggers is similar; we will see an example using SQL:1999 syntax shortly.) The trigger called *init_count* initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called *incr_count* increments the counter for each inserted tuple that satisfies the condition *age* < 18.

One of the example triggers in Figure 5.20 executes before the aetivating statement, and the other example executes after it. A trigger can also be scheduled to execute *instead of* the activating statement; or in *deferred* fashion, at the end of the transaction containing the activating statement; or in *asynchronous* fashion, as part of a separate transaction.

The example in Figure 5.20 illustrates another point about trigger execution: A user must be able to specify whether a trigger is to be executed once per modified record or once per activating statement. If the action depends on individual changed records, for example, we have to examine the *age* field of the inserted Students record to decide whether to increment the count, the trigger-

```
CREATE TRIGGER iniLcount BEFORE INSERT ON Students      /* Event */
    DECLARE
        count INTEGER;
    BEGIN                                               /* Action */
        count := 0;
    END

CREATE TRIGGER incLcount AFTER INSERT ON Students       /* Event */
    WHEN (new.age < 18)    /* Condition; 'new' is just-inserted tuple */
    FOR EACH ROW
    BEGIN        /* Action; a procedure in Oracle's PL/SQL syntax */
        count := count + 1;
    END
```

Figure 5.20 Examples Illustrating Triggers

ing event should be defined to occur for each modified record; the FOR EACH ROW clause is used to do this. Such a trigger is called a **row-level trigger**. On the other hand, the *iniLcount* trigger is executed just once per INSERT statement, regardless of the number of records inserted, because we have omitted the FOR EACH ROW phrase. Such a trigger is called a **statement-level trigger**.

In Figure 5.20, the keyword new refers to the newly inserted tuple. If an existing tuple were modified, the keywords old and new could be used to refer to the values before and after the modification. SQL:1999 also allows the action part of a trigger to refer to the *set* of changed records, rather than just one changed record at a time. For example, it would be useful to be able to refer to the set of inserted Students records in a trigger that executes once after the INSERT statement; we could count the number of inserted records with *age* < 18 through an SQL query over this set. Such a trigger is shown in Figure 5.21 and is an alternative to the triggers shown in Figure 5.20.

The definition in Figure 5.21 uses the syntax of SQL:1999, in order to illustrate the similarities and differences with respect to the syntax used in a typical current DBMS. The keyword clause NEW TABLE enables us to give a table name (InsertedTuples) to the set of newly inserted tuples. The FOR EACH STATEMENT clause specifies a statement-level trigger and can be omitted because it is the default. This definition does not have a WHEN clause; if such a clause is included, it follows the FOR EACH STATEMENT clause, just before the action specification.

The trigger is evaluated once for each SQL statement that inserts tuples into Students, and inserts a single tuple into a table that contains statistics on modifications to database tables. The first two fields of the tuple contain constants (identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with *age* < 18. (The trigger in Figure 5.20 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

```
CREATE TRIGGER seLcount AFTER INSERT ON Students        j* Event *j
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
    INSERT                                              j* Action *j
        INTO StatisticsTable(ModifiedTable, ModificationType, Count)
        SELECT 'Students', 'Insert', COUNT *
        FROM InsertedTuples I
        WHERE 1.age < 18
```

Figure 5.21 Set-Oriented Trigger

5.9 DESIGNING ACTIVE DATABASES

Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution. The effect of a collection of triggers can be very complex, and maintaining an active database can become very difficult. Often, a judicious use of integrity constraints can replace the use of triggers.

5.9.1 Why Triggers Can Be Hard to Understand

In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement. If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part.

If a statement activates more than one trigger, the DBMS typically processes all of them, in senne arbitrary order. An important point is that the execution of the action part of a trigger could in turn activate another trigger. In particular, the execution of the action part of a trigger could again activate the sarne trigger; such triggers are called recursive triggers. The potential for such *chain* activations and the unpredictable order in which a DBMS processes activated triggers can make it difficult to understand the effect of a collection of triggers.

5.9.2 Constraints versus Triggers

A common use of triggers is to maintain database consistency, and in such cases, we should always consider whether using an integrity constraint (e.g., a foreign key constraint) achieves the same goals. The meaning of a constraint is not defined operationally, unlike the effect of a trigger. This property makes a constraint easier to understand, and also gives the DBMS more opportunities to optimize execution. A constraint also prevents the data from being made inconsistent by *any* kind of statement, whereas a trigger is activated by a specific kind of statement (INSERT, DELETE, or UPDATE). Again, this restriction makes a constraint easier to understand.

On the other hand, triggers allow us to maintain database integrity in more flexible ways, as the following examples illustrate.

- Suppose that we have a table called Orders with fields *iternid, quantity, custornerid,* and *unitprice.* When a customer places an order, the first three field values are filled in by the user (in this example, a sales clerk). The fourth field's value can be obtained from a table called Items, but it is important to include it in the Orders table to have a complete record of the order, in case the price of the item is subsequently changed. We can define a trigger to look up this value and include it in the fourth field of a newly inserted record. In addition to reducing the number of fields that the clerk has to type in, this trigger eliminates the possibility of an entry error leading to an inconsistent price in the Orders table.

- Continuing with this example, we may want to perform some additional actions when an order is received. For example, if the purchase is being charged to a credit line issued by the company, we may want to check whether the total cost of the purchase is within the current credit limit. We can use a trigger to do the check; indeed, we can even use a CHECK constraint. Using a trigger, however, allows us to implement more sophisticated policies for dealing with purchases that exceed a credit limit. For instance, we may allow purchases that exceed the limit by no more than 10% if the customer has dealt with the company for at least a year, and add the customer to a table of candidates for credit limit increases.

5.9.3 Other Uses of Triggers

Many potential uses of triggers go beyond integrity maintenance. Triggers can alert users to unusual events (as reflected in updates to the database). For example, we may want to check whether a customer placing an order has made enough purchases in the past month to qualify for an additional discount; if so, the sales clerk must be informed so that he (or she) can tell the customer and possibly generate additional sales! \Ve can relay this information by using a trigger that checks recent purchases and prints a message if the customer qualifies for the discount.

Triggers can generate a log of events to support auditing and security checks. For example, each time a customer places an order, we can create a record with the customer's ID and current credit limit and insert this record in a customer history table. Subsequent analysis of this table might suggest candidates for an increased credit limit (e.g., customers who have never failed to pay a bill on time and who have come within 10% of their credit limit at least three times in the last month).

As the examples in Section 5.8 illustrate, we can use triggers to gather statistics on table accesses and modifications. Some database systems even use triggers internally as the basis for managing replicas of relations (Section 22.11.1). Our list of potential uses of triggers is not exhaustive; for example, triggers have also been considered for workflow management and enforcing business rules.

5.10 REVIEW QUESTIONS

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

- What are the parts of a basic SQL query? Are the input and result tables of an SQL query sets or multisets? How can you obtain a set of tuples as the result of a query? (Section 5.2)
- What are range variables in SQL? How can you give names to output columns in a query that are defined by arithmetic or string expressions? What support does SQL offer for string pattern matching? (Section 5.2)
- What operations does SQL provide over (multi)sets of tuples, and how would you use these in writing queries? (Section 5.3)
- What are nested queries? What is *correlation* in nested queries? How would you use the operators IN, EXISTS, UNIQUE, ANY, and ALL in writing nested queries? Why are they useful? Illustrate your answer by showing how to write the *division* operator in SQL. (Section 5.4)
- What aggregate operators does SQL support? (Section 5.5)
- What is *grouping*? Is there a counterpart in relational algebra? Explain this feature, and disclls the interaction of the HAVING and WHERE clauses. Mention any restrictions that mllst be satisfied by the fields that appear in the GROUP BY clause. (Section 5.5.1)

- \Vhat are *null* values? Are they supported in the relational model, as described in Chapter 3'1 How do they affect the meaning of queries? Can primary key fields of a table contain *null* values? (Section 5.6)
- What types of SQL constraints can be specified using the query language? Can you express primary key constraints using one of these new kinds of constraints? If so, why does SQL provide for a separate primary key constraint syntax? (Section 5.7)
- What is a *trigger*, and what are its three parts? What are the differences between row-level and statement-level triggers? (Section 5.8)
- \Vhy can triggers be hard to understand? Explain the differences between triggers and integrity constraints, and describe when you would use triggers over integrity constrains and vice versa. What are triggers used for? (Section 5.9)

EXERCISES

Online material is available for all exercises in this chapter on the book's webpage at

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

This includes scripts to create tables for each exercise for use with Oracle, IBM DB2, Microsoft SQL Server, and MySQL.

Exercise 5.1 Consider the following relations:

Student(*snum:* integer, *sname:* string, *major:* string, *level:* string, *age:* integer)
Class(*name:* string, *meets_at:* time, *room:* string, *fid:* integer)
Enrolled(*snum:* integer, *cname:* string)
Faculty(*fid:* integer, *fnarne:* string, *deptid:* integer)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

Write the following queries in SQL. No duplicates should be printed in any of the ans\vers.

1. Find the nari1es of all Juniors (level = JR) who are enrolled in a class taught by 1. Teach.
2. Find the age of the oldest student who is either a History major or enrolled in a course taught by I. Teach.
3. Find the names of all classes that either meet in room R128 or have five or more students enrolled.
4. Find the Ilames of all students who are enrolled in two classes that meet at the same time.

5. Find the names of faculty members \vho teach in every room in which some class is taught.
6. Find the names of faculty members for \vhorn the combined enrollment of the courses that they teach is less than five.
7. Print the level and the average age of students for that level, for each level.
8. Print the level and the average age of students for that level, for all levels except JR.
9. For each faculty member that has taught classes only in room *R128,* print the faculty member's name and the total number of classes she or he has taught.
10. Find the names of students enrolled in the maximum number of classes.
11. Find the names of students not enrolled in any class.
12. For each age value that appears in Students, find the level value that appears most often. For example, if there are more FR level students aged 18 than SR, JR, or SO students aged 18, you should print the pair (18, FR).

Exercise 5.2 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. Write the following queries in SQL:

1. Find the *pnames* of parts for which there is some supplier.
2. Find the *snames* of suppliers who supply every part.
3. Find the *snames* of suppliers who supply every red part.
4. Find the *pnamcs* of parts supplied by Acme Widget Suppliers and no one else.
5. Find the *sids* of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).
6. For each part, find the *sname* of the supplier who charges the most for that part.
7. Find the *sids* of suppliers who supply only red parts.
8. Find the *sids* of suppliers who supply a red part anel a green part.
9. Find the *sids* of suppliers who supply a red part or a green part.
10. For every supplier that only supplies green parts, print the name of the supplier and the total number of parts that she supplies.
11. For every supplier that supplies a green part and a reel part, print the name and price of the most expensive part that she supplies.

Exercise 5.3 The following relations keep track of airline flight information:

> Flights(.flno: integer, *from:* string, *to:* string, *di8tance:* integer,
> *departs:* time, *arrives:* time, *price:* integer)
> Aircraft(*aid:* integer, *aname:* string, *cruisingrange:* integer)
> Certified(*eid:* integer, *aid:* integer)
> Employees(*eid:* integer, *ename:* string, *salary:* integer)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft, and only pilots are certified to fly. Write each of the follO\ving queries in SQL. (*Additional queries using the same schema are listed in the exercises for Chapter 4·)*

1. Find the names of aircraft such that all pilots certified to operate them earn more than $80,000.
2. For each pilot who is certified for more than three aircraft, find the *eid* and the maximum *cruisingrange* of the aircraft for which she or he is certified.
3. Find the names of pilots whose *salary* is less than the price of the cheapest route from Los Angeles to Honolulu.
4. For all aircraft with *cruisingrange* over 1000 miles, find the name of the aircraft and the average salary of all pilots certified for this aircraft.
5. Find the names of pilots certified for some Boeing aircraft.
6. Find the *aids* of all aircraft that can be used on routes from Los Angeles to Chicago.
7. Identify the routes that can be piloted by every pilot who makes more than $100,000.
8. Print the *enames* of pilots who can operate planes with *cruisingmnge* greater than 3000 miles but are not certified on any Boeing aircraft.
9. A customer wants to travel from Madison to New York with no more than two changes of flight. List the choice of departure times from Madison if the customer wants to arrive in New York by 6 p.m.
10. Compute the difference between the average salary of a pilot and the average salary of all employees (including pilots).
11. Print the name and salary of every nonpilot whose salary is more than the average salary for pilots.
12. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles.
13. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles, but on at least two such aircrafts.
14. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles and who are certified on some Boeing aircraft.

Exercise 5.4 Consider the following relational schema. An employee can work in more than one department; the *pct_time* field of the Works relation shows the percentage of time that a given employee works in a given department.

> <u>*Emp(eid:*</u> <u>integer,</u> *ename:* string, *age:* integer, *salary:* real)
> <u>Works(eid:</u> <u>integer,</u> <u>*did:*</u> <u>integer,</u> *pet_time:* integer)
> <u>*Dept(did.·*</u> <u>integer,</u> *budget:* real, *managerid:* integer)

Write the following queries in SQL:

1. Print the names and ages of each employee who works in both the Hardware department and the Software department.
2. For each department with more than 20 full-time-equivalent employees (i.e., where the part-time and full-time employees add up to at least that many full-time employees), print the *did* together with the number of employees that work in that department.

sid	*sname*	*rating*	*age*
18	jones	3	30.0
41	jonah	6	56.0
22	ahab	7	44.0
63	moby	*null*	15.0

Figure 5.22 An Instance of Sailors

3. Print the name of each employee whose salary exceeds the budget of all of the departments that he or she works in.
4. Find the *managerids* of managers who manage only departments with budgets greater than $1 million.
5. Find the *enames* of managers who manage the departments with the largest budgets.
6. If a manager manages more than one department, he or she *controls* the sum of all the budgets for those departments. Find the *managerids* of managers who control more than $5 million.
7. Find the *managerids* of managers who control the largest amounts.
8. Find the *enames* of managers who manage only departments with budgets larger than $1 million, but at least one department with budget less than $5 million.

Exercise 5.5 Consider the instance of the Sailors relation shown in Figure 5.22.

1. Write SQL queries to compute the average rating, using AVG; the sum of the ratings, using SUM; and the number of ratings, using COUNT.
2. If you divide the sum just computed by the count, would the result be the same as the average? How would your answer change if these steps were carried out with respect to the *age* field instead of *rating*?
3. Consider the following query: *Find the names of sailors with a higher rating than all sailors with age* < 21. The following two SQL queries attempt to obtain the answer to this question. Do they both compute the result? If not, explain why. Under what conditions would they compute the same result?

```
SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXISTS ( SELECT *
                     FROM   Sailors S2
                     WHERE  S2.age < 21
                            AND S.rating <= S2.rating )
SELECT  *
FROM    Sailors S
WHERE   S.rating > ANY (SELECT  S2.rating
                        FROM    Sailors S2
                        WHERE   S2.age < 21
```

4. Consider the instance of Sailors shown in Figure 5.22. Let us define instance S1 of Sailors to consist of the first two tuples, instance S2 to be the last two tuples, and S to be the given instance.

Show the left outer join of S with itself, with the join condition being *sid=sid*.

(b) Show the right outer join of S with itself, with the join condition being *sid=sid*.

(c) Show the full outer join of S with itself, with the join condition being *S'id=sid*.

(d) Show the left outer join of Sl with S2, with the join condition being *sid=sid*.

(e) Show the right outer join of Sl with S2, with the join condition being *sid=sid*.

(f) Show the full outer join of 81 with S2, with the join condition being *sid=sid*.

Exercise 5.6 Answer the following questions:

1. Explain the term *'impedance mismatch* in the context of embedding SQL commands in a host language such as C.
2. How can the value of a host language variable be passed to an embedded SQL command?
3. Explain the WHENEVER command's use in error and exception handling.
4. Explain the need for cursors.
5. Give an example of a situation that calls for the use of embedded SQL; that is, interactive use of SQL commands is not enough, and some host lang;uage capabilities are needed.
6. Write a C program with embedded SQL commands to address your example in the previous answer.
7. Write a C program with embedded SQL commands to find the standard deviation of sailors' ages.
8. Extend the previous program to find all sailors whose age is within one standard deviation of the average age of all sailors.
9. Explain how you would write a C program to compute the transitive closure of a graph, represented as an 8QL relation Edges(*from, to)*, using embedded SQL commands. (You need not write the program, just explain the main points to be dealt with.)
10. Explain the following terms with respect to cursors: *updatability, sens,itivity,* and *scrollability*.
11. Define a cursor on the Sailors relation that is updatable, scrollable, and returns answers sorted by *age*. Which fields of Sailors can such a cursor *not* update? Why?
12. Give an example of a situation that calls for dynamic 8QL; that is, even embedded SQL is not sufficient.

Exercise 5.7 Consider the following relational schema and briefly answer the questions that follow:

Emp(<u>*eid:* integer</u>, *ename:* string, *age:* integer, *salary:* real)
\Vorks(<u>*eid:* integer</u>, <u>*did:* integer</u>, *pct_time:* integer)
Dept(<u>*did:* integer</u>, *budget:* real, *managerid:* integer)

1. Define a table constraint on Emp that will ensure that every employee makes at least $10,000.
2. Define a table constraint on Dept that will ensure that all managers have *age* > 30.
3. Define an assertion on Dept that will ensure that all managers have *age* > 30. Compare this assertion with the equivalent table constraint. Explain which is better.

4. Write SQL statements to delete all information about employees whose salaries exceed that of the manager of one or more departments that they work in. Be sure to ensure that all the relevant integrity constraints are satisfied after your updates.

Exercise 5.8 Consider the following relations:

> Student(*snum:* integer, *sname:* string, *rnajor:* string,
> *level:* string, *age:* integer)
> Class(narne: string, *meets_at:* time, *roorn:* string, *fid:* integer)
> Enrolled(*snum:* integer, *cnarne:* string)
> Faculty(*fid:* integer, *fnarne:* string, *deptid:* integer)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

1. Write the SQL statements required to create these relations, including appropriate versions of all primary and foreign key integrity constraints.
2. Express each of the following integrity constraints in SQL unless it is implied by the primary and foreign key constraint; if so, explain how it is implied. If the constraint cannot be expressed in SQL, say so. For each constraint, state what operations (inserts, deletes, and updates on specific relations) must be monitored to enforce the constraint.
   (a) Every class has a minimum enrollment of 5 students and a maximum enrollment of 30 students.
   (b) At least one dass meets in each room.
   (c) Every faculty member must teach at least two courses.
   (d) Only faculty in the department with *deptid=33* teach more than three courses.
   (e) Every student must be enrolled in the course called lVlathlOl.
   (f) The room in which the earliest scheduled class (i.e., the class with the smallest *meets_at* value) meets should not be the same as the room in which the latest scheduled class meets.
   (g) Two classes cannot meet in the same room at the same time.
   (h) The department with the most faculty members must have fewer than twice the number of faculty members in the department with the fewest faculty members.
   (i) No department can have more than 10 faculty members.
   (j) A student cannot add more than two courses at a time (i.e., in a single update).
   (k) The number of CS majors must be more than the number of Math majors.
   (l) The number of distinct courses in which CS majors are enrolled is greater than the number of distinct courses in which Math majors are enrolled.
   (rn) The total enrollment in courses taught by faculty in the department with *deptid=33* is greater than the number of ivlath majors.
   (n) There lIlUst be at least one CS major if there are any students whatsoever.
   (0) Faculty members from different departments cannot teach in the same room.

Exercise 5.9 Discuss the strengths and weaknesses of the trigger mechanism. Contrast triggers with other integrity constraints supported by SQL.

Exercise **5.10** Consider the following relational schema. An employee can work in more than one department; the *pct_time* field of the Works relation shows the percentage of time that a given employee works in a given department.

> Emp(<u>*eid:* integer</u>, *ename:* string, *age:* integer, *salary:* real)
> Works(<u>*eid:* integer</u>, <u>*did:* integer</u>, *pct_time:* integer)
> Dept(<u>*did:* integer</u>, *budget:* real, *managerid:* integer)

Write SQL-92 integrity constraints (domain, key, foreign key, or CHECK constraints; or assertions) or SQL:1999 triggers to ensure each of the following requirements, considered independently.

1. Employees must make a minimum salary of $1000.
2. Every manager must be also be an employee.
3. The total percentage of all appointments for an employee must be under 100%.
4. A manager must always have a higher salary than any employee that he or she manages.
5. Whenever an employee is given a raise, the manager's salary must be increased to be at least as much.
6. Whenever an employee is given a raise, the manager's salary must be increased to be at least as much. Further, whenever an employee is given a raise, the department's budget must be increased to be greater than the sum of salaries of all employees in the department.

PROJECT-BASED EXERCISE

Exercise **5.11** Identify the subset of SQL queries that are supported in Minibase.

BIBLIOGRAPHIC NOTES

The original version of SQL was developed as the query language for IBM's System R project, and its early development can be traced in [107, 151]. SQL has since become the most widely used relational query language, and its development is now subject to an international standardization process.

A very readable and comprehensive treatment of SQL-92 is presented by Melton and Simon in [524], and the central features of SQL:1999 are covered in [525]. We refer readers to these two books for an authoritative treatment of SQL. A short survey of the SQL:1999 standard is presented in [237]. Date offers an insightful critique of SQL in [202]. Although some of the problems have been addressed in SQL-92 and later revisions, others remain. A formal semantics for a large subset ofSQL queries is presented in [560]. SQL:1999 is the current International Organization for Standardization (ISO) and American National Standards Institute (ANSI) standard. Melton is the editor of the ANSI and ISO SQL:1999 standard, document ANSI/ISO/IEe 9075-:1999. The corresponding ISO document is ISO/lEe 9075-:1999. A successor, planned for 2003, builds on SQL:1999 SQL:2003 is close to ratification (as of June 20(2). Drafts of the SQL:2003 deliberations are available at the following URL:

> ftp://sqlstandards.org/SC32/

[774] contains a collection of papers that cover the active database field. [794] includes a good in-depth introduction to active rules, covering smnantics, applications and design issues. [251] discusses SQL extensions for specifying integrity constraint checks through triggers. [123] also discusses a procedural mechanism, called an *alerter*, for monitoring a database. [185] is a recent paper that suggests how triggers might be incorporated into SQL extensions. Influential active database prototypes include Ariel [366], HiPAC [516J, ODE [18], Postgres [722], RDL [690], and Sentinel [36]. [147] compares various architectures for active database systems.

[32] considers conditions under which a collection of active rules has the same behavior, independent of evaluation order. Semantics of active databases is also studied in [285] and [792]. Designing and managing complex rule systems is discussed in [60, 225]. [142] discusses rule management using Chimera, a data model and language for active database systems.