It'sIt's your choice!

New Modular Organization!

Applications emphasis:

Appncatirms emphasis: A course that covers the principles of database systems

Systems and emphasizes how they are used in developing data-intensive applications. . f,;-tY'W;Yl~t';;;;,~7'

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

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

you want. ......- :=

Dependencies

~~~

I

II

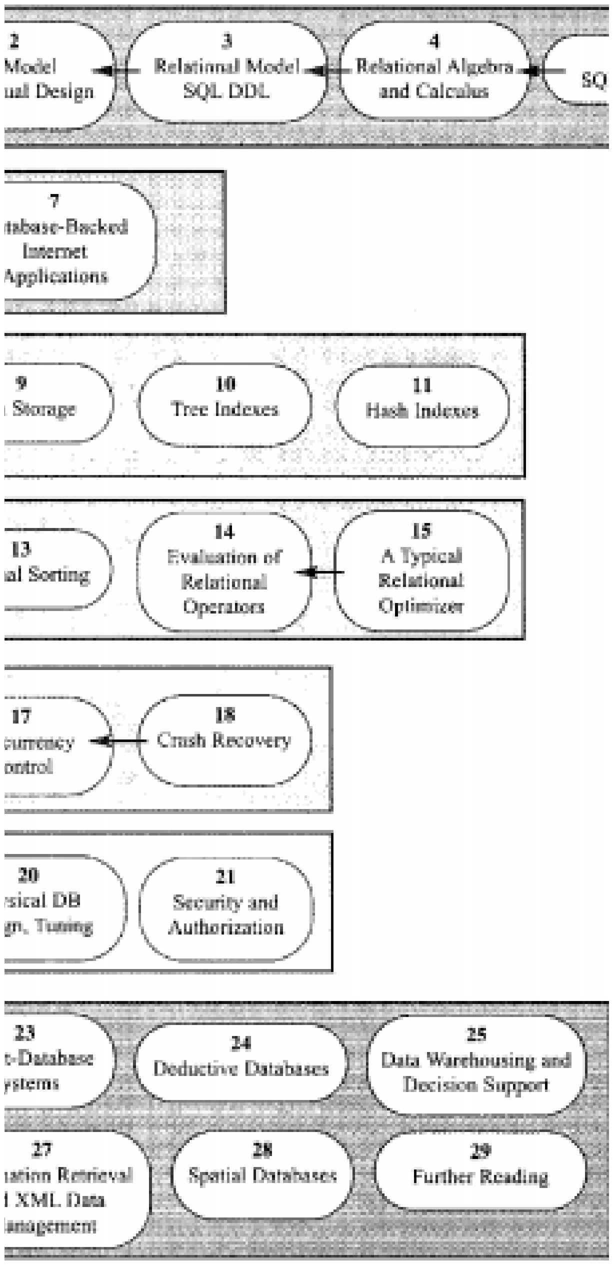
III

IV

V

I

VIr

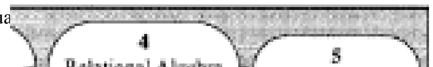


2 Relational Model SQLDDL

ER Model

Conceptua

3


~~~

27
Infonnation Retrieval and XML Data Management

j
j
j
j
j
j
j
j
j
j
j
j
j
j
j
j
j
j
j

DATABASE MANAGEMENT SYSTEMS

DATABASE MANAGEMENT SYSTEMS

Third Edition

Raghu Ramakrishnan
University of Wisconsin
Madison, Wisconsin, USA

●

Johannes Gehrke
Cornell University
Ithaca, New York, USA

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

McGraw-Hill Higher Education tz
A Lhvision 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

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

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 quantita tive approach is used throughout with many detailed examples. An extensive set of exercises (for which solutions are available online to instructors) accom panies each chapter and reinforces students' ability to apply the concepts to real problems.

The book can be used with the accompanying software and programming as signments 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 ap plications. Two new chapters on application development (one on database backed applications, and one on Java and Internet application architec tures) 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.

r..,1any instructors will no doubt teach a course that falls between these two extremes. The restructuring in the third edition offers a very modular orga nization 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 empha..':iis.

The *introductory applications- oriented course* could cover the :Foundations chap ters, then the Application Development chapters, followed by the overview sys tems 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).

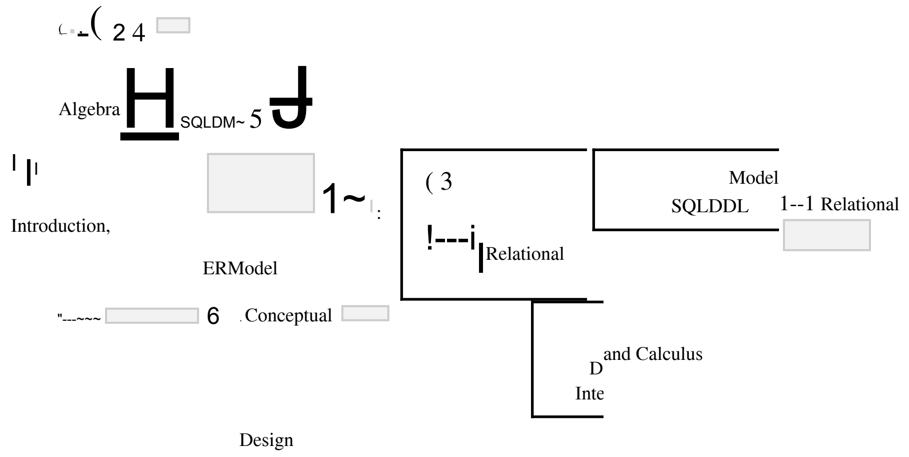
The *introductory systems-oriented course* would cover the Foundations chap ters and a selection of Applications and Systems chapters. An important point for systems-oriented courses is that the timing of programming projects (e.g., using Minibase) makes it desirable to cover some systems topics early. Chap ter 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. Obvi ously, 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 a.spects. 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 a.'3 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.



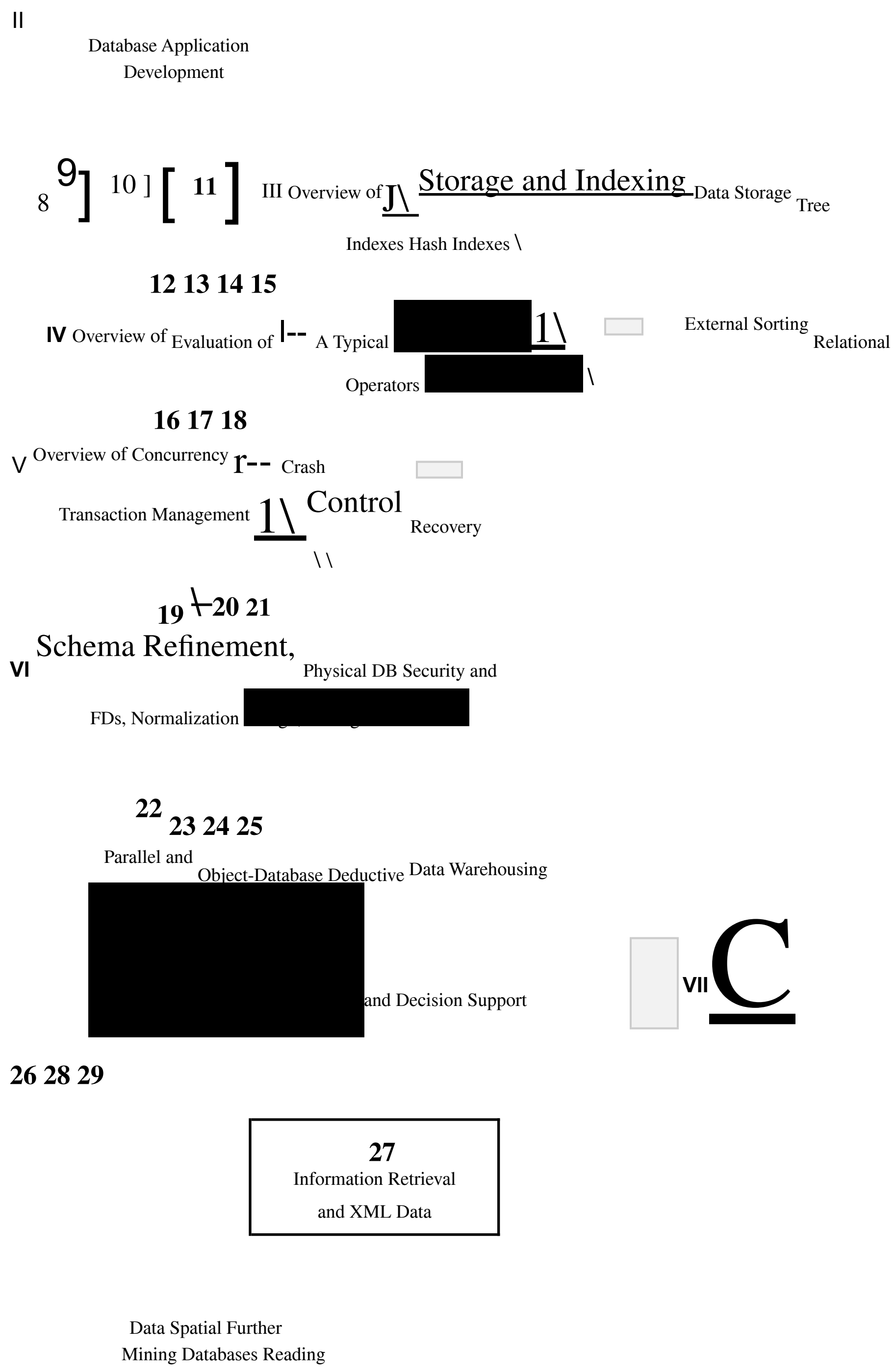

II
Database Application Development
8 9 10 11
III Overview of Storage and Indexing
Data Storage
Tree Indexes
Hash Indexes
12 13 14 15
IV Overview of Evaluation of Relational Operators
A Typical
External Sorting
16 17 18
V Overview of Transaction Management
Concurrency Control
Crash Recovery
19 20 21
VI Schema Refinement, FDs, Normalization
Physical DB
Security and
22 23 24 25
Parallel and
Object-Database
Deductive
Data Warehousing and Decision Support
VII
26 28 29
27
Information Retrieval and XML Data
Data Mining
Spatial Databases
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 teach ing 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 UnUS1H:l,lly extensive set of in-depth exercises. Students can obtain solutioIls to odd-numbered chapter exercises and a set of lecture slides for each chapter through the vVeb 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 J\!Iinibase, a small relational DBMS intended for use in systems-oriented courses. Minibase comes with sample assignments and solutions, as de scribed 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 of fers 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 fre quently updated *link to all known erTOT8 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 compre hensive (e.g., it has a query optimizer and includes visualization software) but

tries to retain the spirit of MinireL lVEke 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 ques tions 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 bibliogra phy 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 vers~on 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 Bekta.'3, 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 com plement 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 1tfcGraw-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 Univer sity, 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 Univer sity, 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 Univer sity 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 Lind say, 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. Zi auddin; 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?" K~tan, 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

\rYe 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 X:NIL. Mike Carey offered input on Web services.

Emily Lupash at McGraw-Hill has been a source of constant support and en couragement. 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 pro
grammer, 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 sci ence 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 inter faces, expert systems and artificial intelligence, statistical techniques, and dynamic programming. \Ve cannot go into all these &<:jpects 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. Oth erwise, 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 man agement systems, with an empha.sis on how to *design* a database and 'li8C a

DBMS effectively. Not surprisingly, many decisions about how to use a DBIvIS for a given application depend on what capabilities the DBMS supports effi ciently. Therefore, to use a DBMS well, it is necessary to also understand how a DBMS *work8.*

Many kinds of database management systems are in use, but this book concen trates on relational database systems (RDBMSs), which are by far the dominant type of DB~IS 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 repos itories, 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 be gin 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 relation ships, 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 ul timately stored is bridged through several *levels of abstract1:on* 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.

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

vVe 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 in fluenced 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 Sys tem 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 *trans actions).* Users write programs a." if they are to be run by themselves, and

the responsibility for running them concurrently is given to the DBlVIS. 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 lan guages 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. Spe cialized 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 ta.'3ks. 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.

Databclse 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 a's 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 Sys tem 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 sys tems could prove to be richly rewarding in more ways than one!

1.3 FILE SYSTEMS VERSUS A DBMS

To understand the need for a DB:~,,1S, let us consider a motivating scenario: A company has a large collection (say, 500 GB[3]) of data on employees, depart ments, 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 crac;hes 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 DBNIS 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 h3..'3 many advantages:

Data Independence: Application programs should not, ideally, be ex posed to details of data representation and storage, The DBJVIS provides an abstract view of the data that hides such details.

Efficient Data Access: A DBMS utilizes a variety of sophisticated tech niques to store and retrieve data efficiently. This feature is especially im pOl'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 contmls* 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 concur rent 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 sup ports 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? Some times, 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 ap plications. 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 datet presented by the DBlVIS does not match the application's needs and actually gets in the way. As an exam ple, relational databa.'3es 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, DBlVISs 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 bi:1Sic constructs, such as the relational model. A databa.se 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. vVe cover the ER model in Chapter 2.

An Example of Poor Design: The relational schema for Students il lustrates 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.

ł*sid* [*name* lZogin

53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan (gmusic	1 1	1.8
53832	Guldu	guldui:Qhnusic	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 rlOt 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.

vVe can make the description of a collection of students more precise by specify ing 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 ar;;pect 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 mar ketplace, the dominant model today is the relational model.

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

1.5.2 Levels **of Abstraction in a DBMS**

The data in a DBMS is described at three levels of abstraction, ar;; 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 coneep tual schemas. \;Ye discuss the DDL facilities of the Inost wid(~ly 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). vVe discuss the three levels of abstraction in the rest of this section.

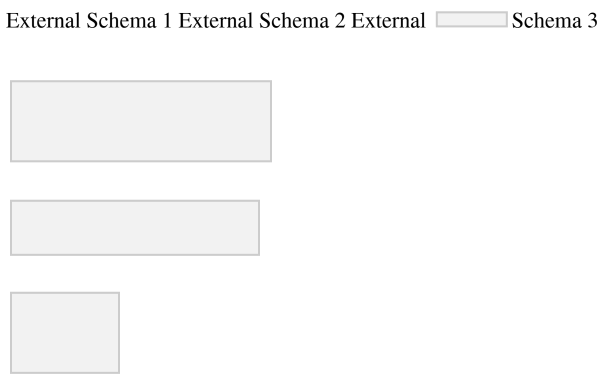


Figure 1.2 Levels of Abstraction in a DBMS

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 databa..'3e, 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(nw: integer, *address:* string, *capacity:* integer) Enrolled (*sid:* string, *cid:* string, *grade:* string)
Teaches(*fid:* string, *cid:* string)
Meets_In(*cid:* string, rno: integer, *ti'fne:* 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. vVe discuss conceptual databa..se 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. \iVe 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, *enTollment:* 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. vVe 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).

L5.3 **Data Independence**

A very important advantage of using a DBMS is that it offers data indepen dence. 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 *(J£d:* 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

4In 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 de termines 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. vVe discuss query optimization and evaluation in Chapters 12, Vl, 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. \Ve 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,. vVe will discuss the DML features of SQL in Chapter 5. The DML and DDL are collectively referred to cl.s 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 look ing up information about available seats oII 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. vVe 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 ta.sk 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 &'3 if they were executing in isolation, one after the other in some order chosen by the DBJ\;:IS. 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 en forced 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 be gins 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* rnod~fy, *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* relea..':les 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 crcLShes just after making the change in the datab(Lse 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 com pleted 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 transac tions 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 databc4'le, and returns the answers. (This is a simplification: SQL commands can be embedded in host-language application programs, e.g., Java or COBOL programs. vVe ignore these issues to concentrate on the core DBl\ilS functionality.)

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

Sophisticated users. application

Unsophisticated users (customers, travel agents, etc.)

Plan Executor

Operator Evaluator

L::::=======~=========::',J'

Engine

Recovery

Manager

DBMS

[C-

Ind,"'I~---""

\ System Catalog

Data Files .--/

Figure 1.3 Architecture of a DBMS

shows references DATABASE

blueprint for evaluating a query, usually represented as a tree of relational op erators (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 dis cussed 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, a:s 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 ct." needed in response to read requests. Buffer management is discussed in Chapter 9.
Ove1'Fie'll} of Database *SY.'3te'171S* 2).

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. DBNIS com ponents associated with concurrency control and recovery include the trans action manager, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transac tions; the lock manager, which keeps track of requests for locks and grants locks on database objects when they become available; and the recovery man ager, 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. Dat,abase imple mentors work for vendors such as IBM or Oracle. End users come from a diverse and increasing number of fields. As data grows in complexity ant(volume, and is increasingly recognized as a major asset, the importance of maintaining it professionally in a DBMS is being widely accepted. Many end user.s simply use applications written by database application programmers (see below) and so require little technical knowledge about DBMS software. Of course, sophisti cated 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 cla.'3ses of people are associated with a DBMS: *application programmer-s* 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 databa'3e is typically maintained by the individual who owns it and uses it. However, corporate or enterprise-wide databases are typically impor tant 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:

- III Design of the Conceptual and Physical Schemas: The DBA is re sponsible 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, al though users probably augment this schema by creating additional views.

- III 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, al though 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.

- III 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).

- I'il 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.

- III vVhat are the main benefits of using a DBMS to manage data in applica tions involving extensive data access? (Sections 1.1, 1.4)

vVhen would you store data in a DBMS instead of in operating system files and vice-versa? (Section 1.3)

- What is a data model? \What is the relational data model? What is data independence and how does a DBNIS support it? (Section 1.5)
- Explain the advantages of using a query language instead of custom pro grams to process data. (Section 1.6)
- What is a transaction? \What guarantees does a DBMS offer with respect to transactions? (Section 1.7)
- What are locks in a DBMS, and why are they used? What is write-ahead logging, and why is it used? What is checkpointing and why is it used? (Section 1.7)
- Identify the main components in a DBMS and briefly explain what they do. (Section 1.8)
- Explain the different roles of database administrators, application program mers, 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 1.1 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 trans actions 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, 3;~9, 475, 574, 689, 747, 762]. [204] provides a detailed discussion of the relational model from a concep tual 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
ing a discussion of the hierar chical and network data models. [339] emphasizes the
een database query languages and logic programming. [762] emphasizes data models. Of
'47] pro vides the most detailed discussion of theoretical issues. Texts devoted to
ects include [3, 45, 501]. Handbook [744] includes a section on databases that contains
ey 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, clataba,se diagrams, component diagrams.

The great successful men of the \vorld 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 (~ntitY-T'd(ltion8hip *(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 databa.'3e design. It provides useful eoncepts that allow us to move fronl an informal description of what users we:mt 1'rorn

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 databa...')e 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.* \Ve 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, how ever, 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 OII.

Database Design Tools: Design tools are available from RDBwiS ven dors 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 require ments 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 dis cussed 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 devel opers 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 straightfor ward 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 databctse 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 dis cuss 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 require ments 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 clatabase design are briefly described next:

4. Schema Refinement: The fourth step ill databa')e design is to analyze the collection of relations in our relational database schema to identify po tential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema re finement can be guided by some elegant and powerful theory. \Ve 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 ad dress 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 ev ery 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 lan guage (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 rnan-

agel' 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). \Ve 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 *s.m.*

2.3 REL~TIONSHIPS 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:

Each n-tuple denotes a relationship involving *n* entities el through *en,* where entity ei 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.

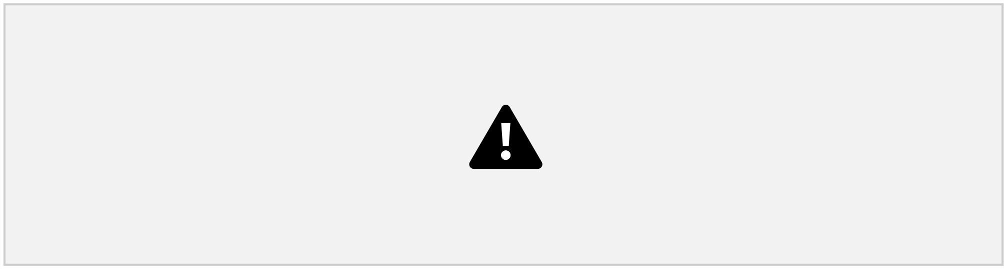
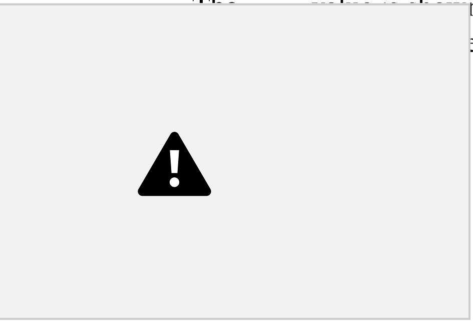


Figure 2.2 The Works-ln Relationship Set

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 At tishoo 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 relation ship 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 em ployee *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 &'3 a 'snapshot' of the relationship set at some instant

in time. An instance of the vVorks.ln 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 n beside each relationship. (The 'many-te-many' and 'total participation' comments ed later, when we discuss integrity constraints.)



__l---\-~---r-~5J

---IILL---t----j-----W

--. __ ---t-----\l-:::::~~

EMPLOYEES

Total participation
WORKS_IN

Many to Many
DEPARTMENTS Total participation

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.

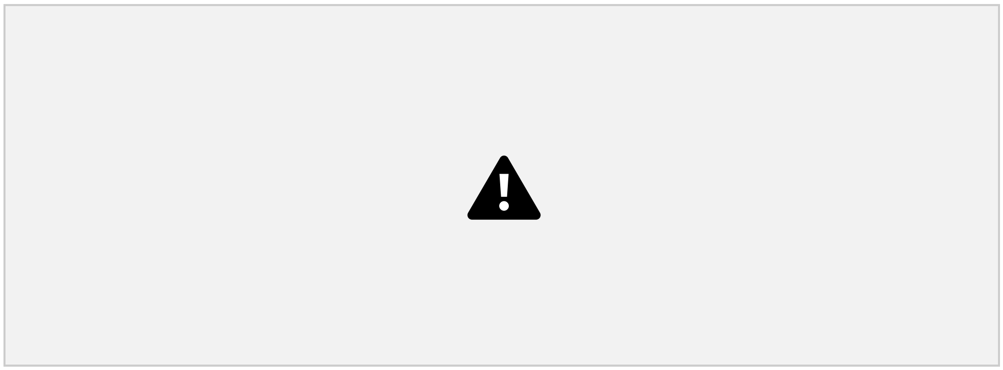


Figure 2.4 A Ternary Relationship Set

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

employees report. to other employees, every relationship in Reports_To is of the form (emlJ1. 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 concate nated 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 relation ship set has attributes corresponding to the *ssn* of the supervisor and the *ssn* of the subordinate, and the names of these attributes are *supcrvisoLssn* and *subordinate-ssn.*

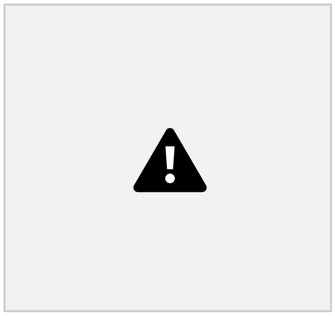


Figure 2.5 The Reports_To Relationship Set

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, &., illustrated in the vVorks_In instance shown in Figure 2.3. Employee 231-31-5368 h&., worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 h&'3 two employees.

Now consider another relationship set called Manages between the Employ ees and Departments entity sets such that each department h&') at most one manager, although a single employee is allowed to manage more than one de partment. The restriction that each department h&,> at most one manager is

an example of a key constraint, and it implies that each Departments entity appears in at most one 1Jlanages relationship in any allowable instance of Man ages. 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 WorksIn relationship set, the instance of Works_In shown in Figure 2.3 violates the key

1123-22-36661.

--..----t-------;'-------a~

! 231-31-53681

[223-32-6316\

EMPLOYEES

Partial participation

MANAGES One to Many

DEPARTMENTS Total participation

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 \Vorks-.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 depl:1J't ment 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 em ploy~e 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 depart ment can be associated with several employees and locations and each location can be associated with several departments and employees; however, each em ployee 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 ha:s at most one manager. A natural question to ask is whether every department ha.'3 a Inan agel'. Let us say that every department is required to have a manager. This requirement is an example of a participation constraint; the particip::ltion 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

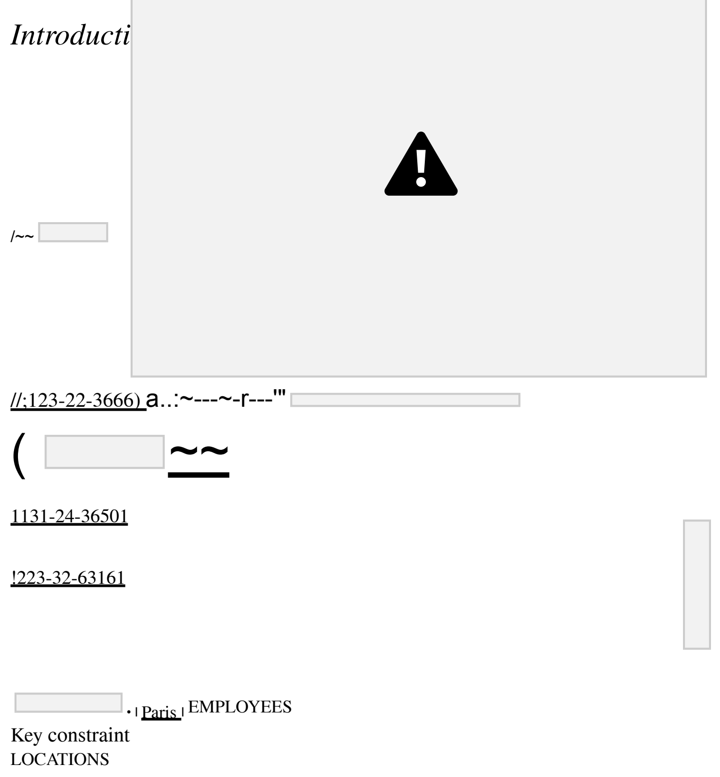


Figure 2.9 An Instance of Works_In3

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 em ployee works in at least one department and that each department has at least one employee. This means that the participation of both Employees and De partments 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 con nected 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. "Ve 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.

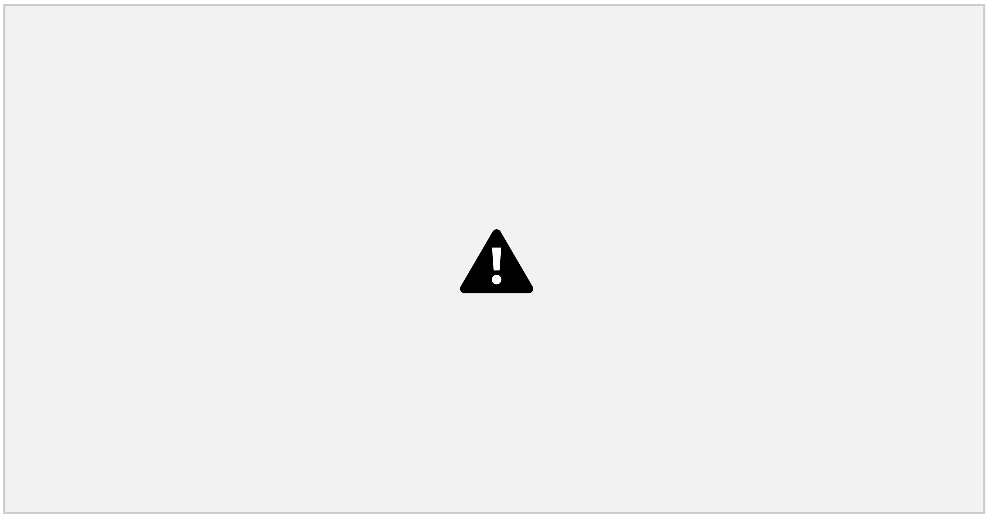


Figure 2.10 Manages and Works_In

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 iden tified 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:

11'I 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.

III The weak entity set must have total participation in the identifying rela tionship 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, be cause 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 Em ployees 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~mps. \Ve 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 &'3 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 cl&ss hierarchy.

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees &'3 SenioLEmps. We can rnodify Figure 2.12 to reflect this change by adding a second ISA node &'3 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.

hourly-wages

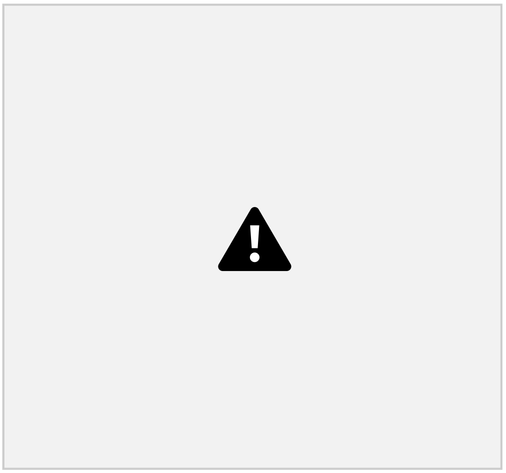


Figure 2.12 Class Hierarchy

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 relation ship sets are then added.

• Hourly -Emps and ContracLEmps are generalized by Employees. As an other example, two entity sets Motorboats and Cars may be generalized into an entity set MotoLVehicles. 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 character istics. 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 At tishoo be both an Hourly_Emps entity and a ContracLEmps entity? Intuitively, no. Can he be both a ContracLEmps entity and a Senior -Emps entity? Intu itively, yes. We denote this by writing 'ContracLE;mps 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 collec tively include all entities in the superclass. For example, does every Employees

entity have to belong to one of its subclasses? Intuitively, no. Does every ~'lotoLVehicles 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. vVe 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 rela tionship. 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 exam ple, 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 Spon sors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Moni tors 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 &'3sociate two or more *entities.*

To define a relationship set such &'3 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 ex press 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 1tntil that records the date until when the employee is appointed as the sponsorship mon itor. 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.

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 &'3 an entity

or a relationship?

II "Vhat arc the relationship sets and their participating entity sets? Should we use binary or ternary relationships?

II 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.

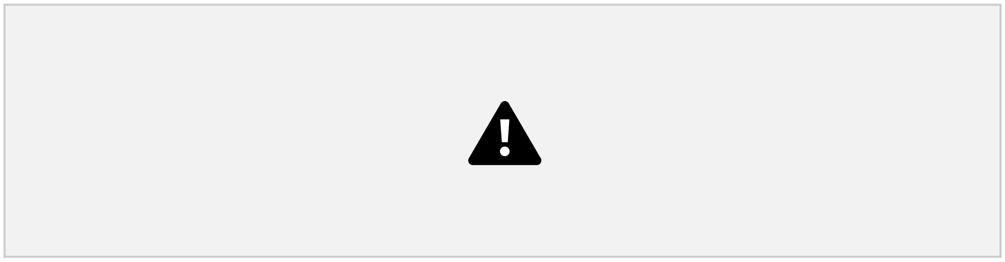


Figure 2.14 The \Vorks_In4 Relationship Set

It has differs from the \Vorks_In relationship set of Figure 2.2 only in that it attributes *JTOtn* 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 rela tionship 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 vVorks-ln2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) vVe can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to,* as shown in Figure 2.15.

(~~-T:~~~)

I

Employees I WorksJn4 Departments

from

to

Figure 2.15 The Works-ln4 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 model ing 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.

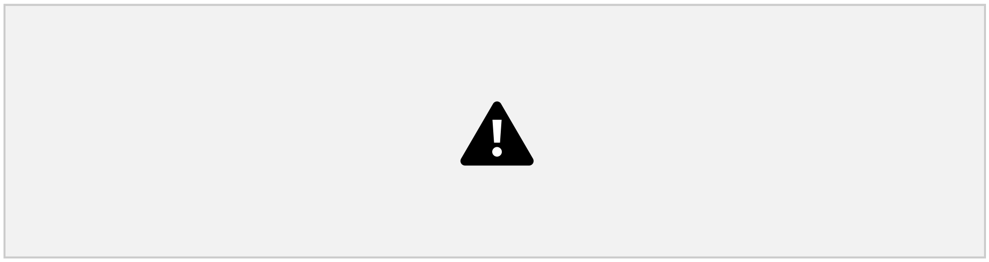


Figure 2.16 Entity versus Relationship

Given a department, we know the manager, as well &'3 the manager's starting date and budget for that department. This approach is natural if we t'l"ssume 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 *db1Ldget* 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.

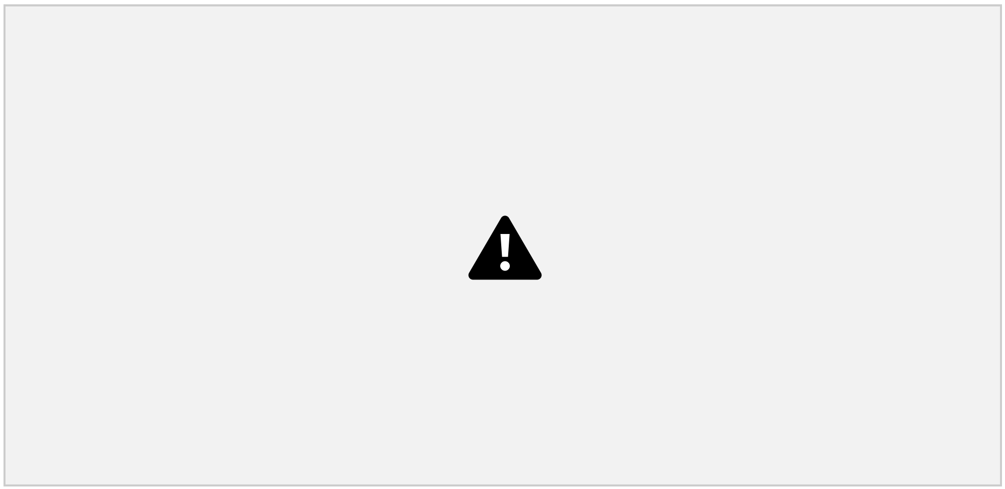


Figure 2.18 Policy Revisited

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, "vhere a relationship inherently a.'3sociates more than two entities. vVe have seen such an example in Figures 2,4 and 2.15.

As a typical example of a ternary relationship, consider entity sets Parts, Sup pliers, 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 depart ment '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 dia gram, 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 *unt-il* 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. VVe 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 &s another reason for using aggregation rather than a ternary relationship set.