FUNDAMENTALS OF DATABASE SYSTEMS

Third Edition



ELMASRI | NAVATHE

Petitioner Uber Ex-1052, 0001

ELMASRI
NAVATHE

FUNDAMENTALS OF DATABASE SYSTEMS

Third Edition

Petitioner Uber Ex-1052, 0002

FUNDAMENTALS OF DATABASE SYSTEMS

Third Edition

Petitioner Uber Ex-1052, 0003

FUNDAMENTALS OF DATABASE SYSTEMS

Third Edition



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

Shamkant B. Navathe
College of Computing
Georgia Institute of Technology

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Menlo Park, California • New York • Harlow, England
Don Mills, Ontario • Sydney • Mexico City • Madrid • Amsterdam

Petitioner Uber Ex-1052, 0004

Acquisitions Editor: Maite Suarez-Rivas
Associate Editor: Katherine Harutunian
Cover Design: Lynne Reed
Composition: Northeast Compositors
Proofreader: Diane Freed

Cover Art: "Crystal Zone"—Original hand-pulled limited edition serigraph by Tetsuro Sawada. Exclusive Sawada publisher and distributor: Buschlen/Mowatt Fine Arts Ltd., Main Floor, 1445 West Georgia Street, Vancouver, Canada V6G 2T3 (604) 682-1234.

Certain diagrams have been reproduced herein with the permission of Oracle Corporation. Copyright © Oracle Corporation, 1997. All rights reserved.

This book was typeset in FrameMaker 5.5 on a Power Macintosh G3. The fonts used were Bodega, Goudy, and Lucida Sans.

Copyright ©2000 by Ramez Elmasri and Shamkant B. Navathe

All rights reserved. No part of this publication may be reproduced, stored in a database or retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or any other media embodiments now known or hereafter to become known, without the prior written permission of the publisher. Printed in the United States of America.

Library of Congress Cataloging-in-Publication Data

Elmasri, Ramez.
Fundamentals of database systems / Ramez Elmasri, Shamkant B. Navathe. – 3rd ed.
p. cm.
Includes bibliographical references.
ISBN 0-8053-1755-4
1. Database management. I. Navathe, Sham. II. Title.
QA76.9.D3E57 2000 99-30620
005.74—dc21 CIP

ISBN 0-8053-1755-4

2 3 4 5 6 7 8 9 10-DOH-0302

Reprinted with corrections, June 2000

Petitioner Uber Ex-1052, 0005

To my wife, Amalia
and to my family.
R. E.

Dedicated to my three greatest teachers,
Father Shri. B. S. Navathe
and grandfathers
Shri. S. N. Navathe and Shri. G. R. Sardesai
and
to my wife Aruna for her infinite patience
S. B. N.

Petitioner Uber Ex-1052, 0006

PREFACE

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

We start in Part 1 with an introduction and a presentation of the basic concepts from both ends of the database spectrum—conceptual modeling principles and physical file storage techniques. We conclude the book in Part 6 with an introduction to influential new database models, such as active, temporal, and deductive models, along with an overview of emerging technologies and applications, such as data mining, data warehousing, and Web databases. Along the way—in Parts 2 through 5—we provide an in-depth treatment of the most important aspects of database fundamentals.

The following key features are included in the third edition:

- The entire book has a self-contained, flexible organization that can be tailored to individual needs.
- Complete and updated coverage is provided on the relational model—including new material on Oracle and Microsoft Access as examples of relational systems—in Part 2.

Petitioner Uber Ex-1052, 0007

- A comprehensive new introduction is provided on object databases and object-relational systems in Part 3, including the ODMG object model and the OQL query language, as well as an overview of object-relational features of SQL3, INFORMIX, and ORACLE 8.
- Updated coverage of EER conceptual modeling has been moved to Chapter 4 to follow the basic ER modeling in Chapter 3, and includes a new section on notation for UML class diagrams.
- Two examples running throughout the book—called COMPANY and UNIVERSITY—allow the reader to compare different approaches that use the same application.
- Coverage has been updated on database design, including conceptual design, normalization techniques, physical design, and database tuning.

The chapters on DBMS system implementation concepts, including catalog, query processing, concurrency control, recovery, and security, now include sections on how these concepts are implemented in real systems.

- New sections with examples on client-server architecture, active databases, temporal databases, and spatial databases have been added.
- There is updated coverage of recent advances in decision support applications of databases, including overviews of data warehousing/OLAP, and data mining.
- State-of-the-art coverage is provided on new database technologies, including Web, mobile, and multimedia databases.
- There is a focus on important new application areas of databases at the turn of the millennium: geographic databases, genome databases, and digital libraries.

Contents of This Edition

Part 1 describes the basic concepts necessary for a good understanding of database design and implementation, as well as the conceptual modeling techniques used in database systems. Chapters 1 and 2 introduce databases, their typical users, and DBMS concepts, terminology, and architecture. In Chapter 3, the concepts of the Entity-Relationship (ER) model and ER diagrams are presented and used to illustrate conceptual database design. Chapter 4 focuses on data abstraction and semantic data modeling concepts, and extends the ER model to incorporate these ideas, leading to the enhanced-ER (EER) data model and EER diagrams. The concepts presented include subclasses, specialization, generalization, and union types (categories). The notation for the class diagrams of UML are also introduced. These are similar to EER diagrams and are used increasingly in conceptual object modeling. Part 1 concludes with a description of the physical file structures and access methods used in database systems. Chapter 5 describes the primary methods of organizing files of records on disk, including static and dynamic hashing. Chapter 6 describes indexing techniques for files, including B-tree and B+-tree data structures and grid files.

Part 2 describes the relational data model and relational DBMSs. Chapter 7 describes

Petitioner Uber Ex-1052, 0008

the basic relational model, its integrity constraints and update operations, and the operations of the relational algebra. Chapter 8 gives a detailed overview of the SQL language, covering the SQL2 standard, which is implemented in most relational systems. Chapter 9 begins with two sections that describe relational schema design, starting from a conceptual database design in an ER or EER model, and concludes with three sections introducing the formal relational calculus languages and an overview of the QBE language. Chapter 10 presents overviews of the Oracle and Microsoft Access database systems as examples of popular commercial relational database management systems.

Part 3 gives a comprehensive introduction to object databases and object-relational systems. Chapter 11 introduces object-oriented concepts and how they apply to object databases. Chapter 12 gives a detailed overview of the ODMG object model and its associated ODL and OQL languages, and gives examples of two commercial object DBMSs. Chapter 13 describes how relational databases are being extended to include object-oriented concepts and presents the features of two object-relational systems—Informix Universal Server and ORACLE 8, as well as giving an overview of some of the features of the proposed SQL3 standard, and the nested relational data model.

Part 4 covers several topics related to database design. Chapters 14 and 15 cover the formalisms, theory, and algorithms developed for relational database design by normalization. This material includes functional and other types of dependencies and normal forms for relations. Step by step intuitive normalization is presented in Chapter 14, and relational design algorithms are given in Chapter 15, which also defines other types of dependencies, such as multivalued and join dependencies. Chapter 16 presents an overview of the different phases of the database design process for medium-sized and large applications, and it also discusses physical database design issues and includes a discussion on database tuning.

Part 5 discusses the techniques used in implementing database management systems. Chapter 17 introduces DBMS system architectures, including centralized and client-server architectures, then describes the system catalog, which is a vital part of any DBMS. Chapter 18 presents the techniques used for processing and optimizing queries specified in a high-level database language—such as SQL—and discusses various algorithms for implementing relational database operations. A section on query optimization in ORACLE has been added. Chapters 19 through 21 discuss transaction processing, concurrency control, and recovery techniques—this material has been revised to include discussions of how these concepts are realized in SQL. Chapter 22 discusses database security and authorization techniques.

Part 6 covers a number of advanced topics. Chapter 23 gives detailed introductions to the concepts of active and temporal databases—which are increasingly being incorporated into database applications—and also gives an overview of spatial and multimedia database concepts. Chapter 24 discusses distributed databases, issues for design, query and transaction processing with data distribution, and the different types of client-server architectures. Chapter 25 introduces the concepts of deductive database systems and surveys a few implementations. Chapter 26 discusses the new technologies of data ware-

Petitioner Uber Ex-1052, 0009

housing and data mining for decision support applications. Chapter 27 surveys the new trends in database technology including Web, mobile and multimedia databases and overviews important emerging applications of databases: geographic information systems (GIS), human genome databases, and digital libraries.

Appendix A gives a number of alternative diagrammatic notations for displaying a conceptual ER or EER schema. These may be substituted for the notation we use, if the instructor so wishes. Appendix B gives some important physical parameters of disks. Appendixes C and D cover legacy database systems, based on the network and hierarchical database models. These have been used for over 30 years as a basis for many existing commercial database applications and transaction-processing systems and will take decades to replace completely. We consider it important to expose students of database management to these long-standing approaches. Full chapters from the second edition can be found at the Website for this edition.

Guidelines for Using This Book

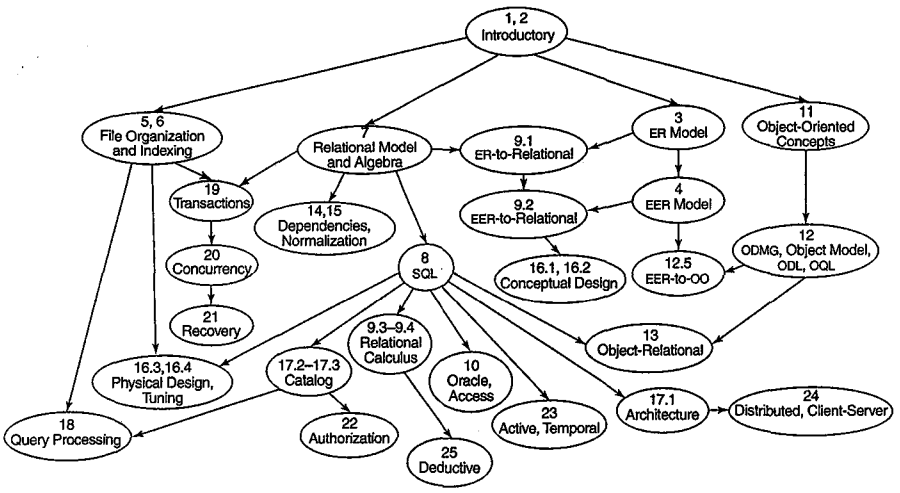
There are many different ways to teach a database course. The chapters in Parts 1 through 3 can be used in an introductory course on database systems in the order they are given or in the preferred order of each individual instructor. Selected chapters and sections may be left out, and the instructor can add other chapters from the rest of the book, depending on the emphasis of the course. At the end of each chapter's opening section, we list sections that are candidates for being left out whenever a less detailed discussion of the topic in a particular chapter is desired. We suggest covering up to Chapter 14 in an introductory database course and including selected parts of Chapters 11 through 13, depending on the background of the students and the desired coverage of the object model. For an emphasis on system implementation techniques, selected chapters from Part 5 can be included. For an emphasis on database design, further chapters from Part 4 can be used.

Chapters 3 and 4, which cover conceptual modeling using the ER and EER models, are important for a good conceptual understanding of databases. However, they may be partially covered, covered later in a course, or even left out if the emphasis is on DBMS implementation. Chapters 5 and 6 on file organizations and indexing may also be covered early on, later, or even left out if the emphasis is on database models and languages. For students who have already taken a course on file organization, parts of these chapters could be assigned as reading material or some exercises may be assigned to review the concepts.

Chapters 10 and 13 include material specific to commercial relational database management systems (RDBMSs)—ORACLE, Microsoft Access, and Informix. Because of the constant revision of these products, no exercises have been assigned in these chapters. Depending on local availability of RDBMSs, material from these chapters may be used in projects. A total life-cycle database design and implementation project covers conceptual design (Chapters 3 and 4), data model mapping (Chapter 9), normalization (Chapter 14), and implementation in SQL (Chapter 8). Additional documentation on the specific RDBMS would be required.

Petitioner Uber Ex-1052, 0010

The book has been written so that it is possible to cover topics in a variety of orders. The chart included here shows the major dependencies between chapters. As the diagram illustrates, it is possible to start with several different topics following the first two introductory chapters. Although the chart may seem complex, it is important to note that if the chapters are covered in order, the dependencies are not lost. The chart can be consulted by instructors wishing to use an alternative order of presentation.



For a single-semester course based on this book, some chapters can be assigned as reading material. Chapters 5, 6, 16, 17, 26, and 27 can be considered for such an assignment. The book can also be used for a two-semester sequence. The first course, "Introduction to Database Design/Systems," at the sophomore, junior, or senior level, could cover most of Chapters 1 to 15. The second course, "Database Design and Implementation Techniques," at the senior or first-year graduate level, can cover Parts 4 through 6. Chapters from Part 6 can be used selectively in either semester, and material describing the DBMS available to the students at the local institution can be covered in addition to the material in the book. Part 6 can also serve as introductory material for advanced database courses, in conjunction with additional assigned readings.

Acknowledgments

It is a great pleasure for us to acknowledge the assistance and contributions of a large number of individuals to this effort. First, we would like to thank our editors, Maite Suarez-Rivas, Katherine Harutunian, Patricia Unubun, and Bob Woodbury. In particular we would like to acknowledge the efforts and help of Katherine Harutunian, our primary contact for the third edition. We would like to acknowledge also those persons who have contributed to the third edition and suggested various improvements to the second edi-

Petitioner Uber Ex-1052, 0011

tion. Suzanne Dietrich wrote parts of Chapter 10 and 12, and Ed Omiecinski contributed to Chapters 17–21. We appreciated the contributions of the following reviewers: François Bançilhon, Jose Blakeley, Rick Cattell, Suzanne Dietrich, David W. Embley, Henry A. Etlinger, Leonidas Fegaras, Farshad Fotouhi, Michael Franklin, Goetz Graefe, Richard Hull, Sushil Jajodia, Ramesh K. Karne, Vijay Kumar, Tarcisio Lima, Ramon A. Mata-Toledo, Dennis McLeod, Rokia Missaoui, Ed Omiecinski, Joan Peckham, Betty Salzberg, Ming-Chien Shan, Junping Sun, Rajshekhar Sunderraman, and Emilia E. Villarreal. In particular, Henry A. Etlinger, Leonidas Fegaras, and Emilla E. Villareal reviewed the entire book.

Sham Navathe would like to acknowledge the substantial contributions of his students Sreejith Gopinath (Chapters 10, 24), Harish Kotbagi (Chapter 25), Jack McCaw (Chapters 26, 27), and Magdi Morsi (Chapter 13). Help on this revision from Rafi Ahmed, Ann Chervenak, Dan Forsyth, M. Narayanaswamy, Carlos Ordonez, and Aravindan Veerasamy has been valuable. Gwen Baker, Amol Navathe, and Aditya Nawathe helped with the manuscript in many ways. Ramez Emasri would like to thank Katrina, Riyad, and Thomas Elmasri for their help with the index and his students at the University of Texas for their comments on the manuscript. We would also like to acknowledge the students at the University of Texas at Arlington and the Georgia Institute of Technology who used drafts of the new material in the third edition.

We would like to repeat our thanks to those who have reviewed and contributed to both previous editions of *Fundamentals of Database Systems*. For the first edition these individuals include Alan Apt (editor), Don Batory, Scott Downing, Dennis Heimbigner, Julia Hodges, Yannis Ioannidis, Jim Larson, Dennis McLeod, Per-Ake Larson, Rahul Patel, Nicholas Roussopoulos, David Stemple, Michael Stonebraker, Frank Tompa, and Kyu-Young Whang; for the second edition they include Dan Joraanstad (editor), Rafi Ahmed, Antonio Albano, David Beech, Jose Blakeley, Panos Chrysanthis, Suzanne Dietrich, Vic Ghorpadey, Goetz Graefe, Eric Hanson, Junguk L. Kim, Roger King, Vram Kouramajian, Vijay Kumar, John Lowther, Sanjay Manchanda, Toshimi Minoura, Inderpal Mumick, Ed Omiecinski, Girish Pathak, Raghu Ramakrishnan, Ed Robertson, Eugene Sheng, David Stotts, Marianne Winslett, and Stan Zdonick.

Last but not least, we gratefully acknowledge the support, encouragement, and patience of our families.

R.E.
S.B.N.

Petitioner Uber Ex-1052, 0012

ABOUT THE AUTHORS

Ramez A. Elmasri is a professor in the department of Computer Science and Engineering at the University of Texas at Arlington. Professor Elmasri previously worked for Honeywell and the University of Houston. He has been an associate editor of the *Journal of Parallel and Distributed Databases* and a member of the steering committee for the International Conference on Conceptual Modeling. He was program chair of the 1993 International Conference on Entity Relationship Approach. He has conducted research sponsored by grants from NSF, NASA, ARRI, Texas Instruments, Honeywell, Digital Equipment Corporation, and the State of Texas in many areas of database systems and in the area of integration of systems and software over the past twenty years. Professor Elmasri has received the Robert Q. Lee teaching award of the College of Engineering of the University of Texas at Arlington. He holds a Ph.D. from Stanford University and has over 70 refereed publications in journals and conference proceedings.

Shamkant Navathe is a professor and the head of the database research group in the College of Computing at the Georgia Institute of Technology. Professor Navathe has previously worked with IBM and Siemens in their research divisions and has been a consultant to various companies including Digital Equipment Corporation, Hewlett-Packard, and Equifax. He has been an associate editor of ACM *Computing Surveys* and IEEE *Transactions on Knowledge and Data Engineering*, and is currently on the editorial boards of *Information Systems* (Pergamon Press) and *Distributed and Parallel Databases* (Kluwer Academic Publishers). He is the co-author of *Conceptual Design: An Entity Relationship Approach* (Addison-Wesley, 1992) with Carlo Batini and Stefano Ceri. Professor Navathe holds a Ph.D. from the University of Michigan and has over 100 refereed publications in journals and conference proceedings.

Petitioner Uber Ex-1052, 0013

CONTENTS

PART 1: BASIC CONCEPTS

Chapter 1 Databases and Database Users 2

1.1 Introduction 4
1.2 An Example 5
1.3 Characteristics of the Database Approach 8
1.4 Actors on the Scene 12
1.5 Workers behind the Scene 13
1.6 Advantages of Using a DBMS 14
1.7 Implications of the Database Approach 18
1.8 When Not to Use a DBMS 19
1.9 Summary 19
Review Questions 20
Exercises 21
Selected Bibliography 21

Chapter 2 Database System Concepts and Architecture 23

2.1 Data Models, Schemas, and Instances 24
2.2 DBMS Architecture and Data Independence 27
2.3 Database Languages and Interfaces 30
2.4 The Database System Environment 32

Petitioner Uber Ex-1052, 0014

2.5 Classification of Database Management Systems 35
2.6 Summary 37
Review Questions 38
Exercises 39
Selected Bibliography 39

Chapter 3 Data Modeling Using the Entity-Relationship Model 41

3.1 Using High-Level Conceptual Data Models for Database Design 42
3.2 An Example Database Application 44
3.3 Entity Types, Entity Sets, Attributes, and Keys 45
3.4 Relationships, Relationship Types, Roles, and Structural Constraints 52
3.5 Weak Entity Types 59
3.6 Refining the ER Design for the COMPANY Database 60
3.7 ER Diagrams, Naming Conventions, and Design Issues 61
3.8 Summary 66
Review Questions 67
Exercises 67
Selected Bibliography 72

Chapter 4 Enhanced Entity-Relationship and Object Modeling 73

4.1 Subclasses, Superclasses, and Inheritance 74
4.2 Specialization and Generalization 76
4.3 Constraints and Characteristics of Specialization and Generalization 80
4.4 Modeling of UNION Types Using Categories 86
4.5 An Example UNIVERSITY EER Schema and Formal Definitions for the EER Model 90
4.6 Conceptual Object Modeling Using UML Class Diagrams 93

Petitioner Uber Ex-1052, 0015

4.7 Relationship Types of Degree Higher Than Two 95
4.8 Data Abstraction and Knowledge Representation Concepts 100
4.9 Summary 105
Review Questions 105
Exercises 106
Selected Bibliography 110

Chapter 5 Record Storage and Primary File Organizations 113

5.1 Introduction 114
5.2 Secondary Storage Devices 117
5.3 Parallelizing Disk Access Using RAID Technology 122
5.4 Buffering of Blocks 127
5.5 Placing File Records on Disk 128
5.6 Operations on Files 133
5.7 Files of Unordered Records (Heap Files) 135
5.8 Files of Ordered Records (Sorted Files) 136
5.9 Hashing Techniques 139
5.10 Other Primary File Organizations 148
5.11 Summary 149
Review Questions 149
Exercises 150
Selected Bibliography 153

Chapter 6 Index Structures for Files 155

6.1 Types of Single-Level Ordered Indexe 156
6.2 Multilevel Indexes 166
6.3 Dynamic Multilevel Indexes Using B-Trees and B^+-Trees 169
6.4 Indexes on Multiple Keys 183
6.5 Other Types of Indexes 186
6.6 Summary 187
Review Questions 188

Petitioner Uber Ex-1052, 0016

Exercises 188
Selected Bibliography 190

PART 2: RELATIONAL MODEL, LANGUAGES, AND SYSTEMS 193

Chapter 7 The Relational Data Model, Relational Constraints, and the Relational Algebra 195

7.1 Relational Model Concepts 196
7.2 Relational Constraints and Relational Database Schemas 202
7.3 Update Operations and Dealing with Constraint Violations 209
7.4 Basic Relational Algebra Operations 211
7.5 Additional Relational Operations 226
7.6 Examples of Queries in Relational Algebra 230
7.7 Summary 232
Review Questions 233
Exercises 235
Selected Bibliography 240

Chapter 8 SQL–The Relational Database Standard 243

8.1 Data Definition, Constraints, and Schema Changes in SQL2 245
8.2 Basic Queries in SQL 251
8.3 More Complex SQL Queries 261
8.4 Insert, Delete, and Update Statements in SQL 275
8.5 Views (Virtual Tables) in SQL 278
8.6 Specifying General Constraints as Assertion 282
8.7 Additional Features of SQL 283
8.8 Summary 284

Petitioner Uber Ex-1052, 0017

Review Questions 284
Exercises 286
Selected Bibliography 288

Chapter 9 ER- and EER-to-Relational Mapping, and Other Relational Languages 289

9.1 Relational Database Design Using ER-to-Relational Mapping 290
9.2 Mapping EER Model Concepts to Relations 295
9.3 The Tuple Relational Calculus 299
9.4 The Domain Relational Calculus 308
9.5 Overview of the QBE Language 310
9.6 Summary 317
Review Questions 318
Exercises 319
Selected Bibliography 320

Chapter 10 Examples of Relational Database Management Systems: Oracle and Microsoft Access 323

10.1 Relational Database Management Systems: A Historical Perspective 324
10.2 The Basic Structure of the Oracle System 325
10.3 Database Structure and Its Manipulation in Oracle 329
10.4 Storage Organization in Oracle 333
10.5 Programming Oracle Applications 337
10.6 Oracle Tools 343
10.7 An Overview of Microsoft Access 344
10.8 Features and Functionality of Access 351
10.9 Summary 355
Selected Bibliography 355

Petitioner Uber Ex-1052, 0018

PART 3: OBJECT-ORIENTED AND EXTENDED RELATIONAL DATABASE TECHNOLOGY 357

Chapter 11 Concepts for Object-Oriented Databases 359

11.1 Overview of Object-Oriented Concepts 361
11.2 Object Identity, Object Structure, and Type Constructors 363
11.3 Encapsulation of Operations, Methods, and Persistence 369
11.4 Type Hierarchies and Inheritance 374
11.5 Complex Objects 377
11.6 Other Object-Oriented Concepts 379
11.7 Summary 382
Review Questions 383
Exercises 384
Selected Bibliography 384

Chapter 12 Object Database Standards, Languages, and Design 385

12.1 Overview of the Object Model of ODMG 386
12.2 The Object Definition Language 399
12.3 The Object Query Language 404
12.4 Overview of the C++ Language Binding 413
12.5 Object Database Conceptual Design 414
12.6 Examples of ODBMSs 417
12.7 Overview of the CORBA Standard for Distributed Objects 428
12.8 Summary 431
Review Questions 432
Exercises 432
Selected Bibliography 433

Petitioner Uber Ex-1052, 0019

Chapter 13 Object Relational and Extended Relational Database Systems 435

13.1 Evolution and Current Trends of Database Technology 436
13.2 The Informix Universal Server 438
13.3 Object-Relational Features of Oracle 8 448
13.4 An Overview of SQL3 451
13.5 Implementation and Related Issues for Extended Type Systems 458
13.6 The Nested Relational Data Model 459
13.7 Summary 461
Selected Bibliography 462

PART 4: DATABASE DESIGN THEORY AND METHODOLOGY 463

Chapter 14 Functional Dependencies and Normalization for Relational Databases 465

14.1 Informal Design Guidelines for Relation Schemas 467
14.2 Functional Dependencies 476
14.3 Normal Forms Based on Primary Keys 483
14.4 General Definitions of Second and Third Normal Forms 490
14.5 Boyce-Codd Normal Form 493
14.6 Summary 496
Review Questions 496
Exercises 497
Selected Bibliography 500

Chapter 15 Relational Database Design Algorithms and Further Dependencies 501

15.1 Algorithms for Relational Database Schema Design 502

Petitioner Uber Ex-1052, 0020

15.2 Multivalued Dependencies and Fourth Normal Form 514
15.3 Join Dependencies and Fifth Normal Form 519
15.4 Inclusion Dependencies 520
15.5 Other Dependencies and Normal Forms 521
15.6 Summary 523
Review Questions 524
Exercises 524
Selected Bibliography 525

Chapter 16 Practical Database Design and Tuning 527

16.1 The Role of Information Systems in Organizations 528
16.2 The Database Design Process 532
16.3 Physical Database Design in Relational Databases 551
16.4 An Overview of Database Tuning in Relational Systems 554
16.5 Automated Design Tools 560
16.6 Summary 563
Review Questions 564
Selected Bibliography 565

PART 5: SYSTEM IMPLEMENTATION TECHNIQUES 568

Chapter 17 Database System Architectures and the System Catalog 569

17.1 System Architectures for DBMSs 570
17.2 Catalogs for Relational DBMSs 573
17.3 System Catalog Information in ORACLE 577
17.4 Other Catalog Information Accessed by DBMS Software Modules 581
17.5 Data Dictionary and Data Repository Systems 582
17.6 Summary 583

Petitioner Uber Ex-1052, 0021

Review Questions 584
Exercises 584
Selected Bibliography 585

Chapter 18 Query Processing and Optimization 585

18.1 Translating SQL Queries into Relational Algebra 587
18.2 Basic Algorithms for Executing Query Operations 588
18.3 Using Heuristics in Query Optimization 604
18.4 Using Selectivity and Cost Estimates in Query Optimization 615
18.5 Overview of Query Optimization in ORACLE 624
18.6 Semantic Query Optimization 625
18.7 Summary 626
Review Questions 626
Exercises 627
Selected Bibliography 628

Chapter 19 Transaction Processing Concepts 629

19.1 Introduction to Transaction Processing 630
19.2 Transaction and System Concepts 637
19.3 Desirable Properties of Transactions 640
19.4 Schedules and Recoverability 641
19.5 Serializability of Schedules 644
19.6 Transaction Support in SQL 654
19.7 Summary 656
Review Questions 657
Exercises 657
Selected Bibliography 659

Chapter 20 Concurrency Control Techniques 661

20.1 Locking Techniques for Concurrency Control 662
20.2 Concurrency Control Based on Timestamp Ordering 672

Petitioner Uber Ex-1052, 0022

20.3 Multiversion Concurrency Control Techniques 674
20.4 Validation (Optimistic) Concurrency Control Techniques 677
20.5 Granularity of Data Items and Multiple Granularity Locking 678
20.6 Using Locks for Concurrency Control in Indexes 683
20.7 Some Other Concurrency Control Issues 684
20.8 Summary 685
Review Questions 686
Exercises 687
Selected Bibliography 687

Chapter 21 Database Recovery Techniques 689

21.1 Recovery Concepts 690
21.2 Recovery Techniques Based on Deferred Update 696
21.3 Recovery Techniques Based on Immediate Update 701
21.4 Shadow Paging 702
21.5 The ARIES Recovery Algorithm 704
21.6 Recovery in Multidatabase Systems 708
21.7 Database Backup and Recovery from Catastrophic Failures 709
21.8 Summary 709
Review Questions 710
Exercises 711
Selected Bibliography 713

Chapter 22 Database Security and Authorization 715

22.1 Introduction to Database Security Issues 715
22.2 Discretionary Access Control Based on Granting/Revoking of Privileges 718
22.3 Mandatory Access Control for Multilevel Security 723
22.4 Introduction to Statistical Database Security 726

Petitioner Uber Ex-1052, 0023

22.5 Summary 728
Review Questions 728
Exercises 729
Selected Bibliography 729

PART 6: ADVANCED DATABASE CONCEPTS & EMERGING APPLICATION 731

Chapter 23 Enhanced Data Models for Advanced Applications 733

23.1 Active Database Concepts 734
23.2 Temporal Database Concepts 744
23.3 Spatial and Multimedia Databases 757
23.4 Summary 760
Review Questions 761
Exercises 762
Selected Bibliography 763

Chapter 24 Distributed Databases and Client-Server Architecture 765

24.1 Distributed Database Concepts 766
24.2 Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design 771
24.3 Types of Distributed Database Systems 778
24.4 Query Processing in Distributed Databases 781
24.5 Overview of Concurrency Control and Recovery in Distributed Databases 786
24.6 An Overview of Client-Server Architecture and Its Relationship to Distributed Databases 789
24.7 Distributed Databases in Oracle 791
24.8 Future Prospects of Client-Server Technology 793
24.9 Summary 795

Petitioner Uber Ex-1052, 0024

Review Questions 795
Exercises 796
Selected Bibliography 798

Chapter 25 Deductive Databases 801

25.1 Introduction to Deductive Databases 802
25.2 Prolog/Datalog Notation 803
25.3 Interpretation of Rules 808
25.4 Basic Inference Mechanisms for Logic Programs 810
25.5 Datalog Programs and Their Evaluation 813
25.6 Deductive Database Systems 827
25.7 Deductive Object-Oriented Databases 832
25.8 Applications of Commercial Deductive Database Systems 834
25.9 Summary 835
Exercises 836
Selected Bibliography 839

Chapter 26 Data Warehousing and Data Mining 841

26.1 Data Warehousing 842
26.2 Data Mining 855
26.3 Summary 870
Review Exercises 871
Selected Bibliography 872

Chapter 27 Emerging Database Technologies and Applications 73

27.1 Databases on the World Wide Web 874
27.2 Multimedia Databases 880
27.3 Mobile Databases 886
27.4 Geographic Information Systems 891
27.5 Genome Data Management 898
27.6 Digital Libraries 905

Petitioner Uber Ex-1052, 0025

Appendix A Alternative Diagrammatic Notations 909

Appendix B Parameters of Disks 913

Appendix C An Overview of the Network Data Model 917

Appendix D An Overview of the Hierarchical Data Model 941

Selected Bibliography B-1

Index I-1

Petitioner Uber Ex-1052, 0026

PART 1

Basic Concepts

Petitioner Uber Ex-1052, 0027

CHAPTER 1 Databases and Database Users

Databases and database systems have become an essential component of everyday life in modern society. In the course of a day, most of us encounter several activities that involve some interaction with a database. For example, if we go to the bank to deposit or withdraw funds; if we make a hotel or airline reservation; if we access a computerized library catalog to search for a bibliographic item; or if we order a magazine subscription from a publisher, chances are that our activities will involve someone accessing a database. Even purchasing items from a supermarket nowadays in many cases involves an automatic update of the database that keeps the inventory of supermarket items.

The above interactions are examples of what we may call **traditional database applications,** where most of the information that is stored and accessed is either textual or numeric. In the past few years, advances in technology have been leading to exciting new applications of database systems. **Multimedia databases** can now store pictures, video clips, and sound messages. **Geographic information systems (GIS)** can store and analyze maps, weather data, and satellite images. **Data warehouses** and **on-line analytical processing (OLAP)** systems are used in many companies to extract and analyze useful information from very large databases for decision making. **Real-time** and **active database technology** is used in controlling industrial and manufacturing processes. And database search techniques are being applied to the World Wide Web to improve the search for information that is needed by users browsing through the Internet.

To understand the fundamentals of database technology, however, we must start from the basics of traditional database applications. So, in Section 1.1 of this chapter we define

Petitioner Uber Ex-1052, 0028

what a database is, and then we give definitions of other basic terms. In Section 1.2, we provide a simple UNIVERSITY database example to illustrate our discussion. Section 1.3 describes some of the main characteristics of database systems, and Sections 1.4 and 1.5 categorize the types of personnel whose jobs involve using and interacting with database systems. Sections 1.6, 1.7, and 1.8 offer a more thorough discussion of the various capabilities provided by database systems and of the implications of using the database approach. Section 1.9 summarizes the chapter.

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

1.1 Introduction

Databases and database technology are having a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and library science, to name a few. The word *database* is in such common use that we must begin by defining a database. Our initial definition is quite general.

A **database** is a collection of related data.[1] By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL. This is a collection of related data with an implicit meaning and hence is a database.

The preceding definition of *database* is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term *database* is usually more restricted. A database has the following implicit properties:

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

In other words, a database has some source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database.

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

Petitioner Uber Ex-1052, 0029

A database can be of any size and of varying complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the card catalog of a large library may contain half a million cards stored under different categories—by primary author's last name, by subject, by book title—with each category organized in alphabetic order. A database of even greater size and complexity is maintained by the Internal Revenue Service to keep track of the tax forms filed by U.S. taxpayers. If we assume that there are 100 million taxpayers and if each taxpayer files an average of five forms with approximately 200 characters of information per form, we would get a database of $100*(10^6)*200*5$ characters (bytes) of information. If the IRS keeps the past three returns for each taxpayer in addition to the current return, we would get a database of $4*(10^{11})$ bytes (400 gigabytes). This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed.

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

A **database management system** (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, and *manipulating* databases for various applications. **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

It is not necessary to use general-purpose DBMS software to implement a computerized database. We could write our own set of programs to create and maintain the database, in effect creating our own *special-purpose* DBMS software. In either case—whether we use a general-purpose DBMS or not—we usually have to employ a considerable amount of software to manipulate the database. We will call the database and DBMS software together a **database system.** Figure 1.1 illustrates these ideas.

1.2 An Example

Let us consider an example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1.2 shows the database structure and a few sample data for such a database. The database is organized as five files, each of which stores data records of the same type.[2] The STUDENT file stores data on each student; the COURSE file stores data on each course; the SECTION file stores data on each section of a course; the GRADE_REPORT file stores

2. At a conceptual level, a *file* is a *collection* of records that may or may not be ordered.

Petitioner Uber Ex-1052, 0030

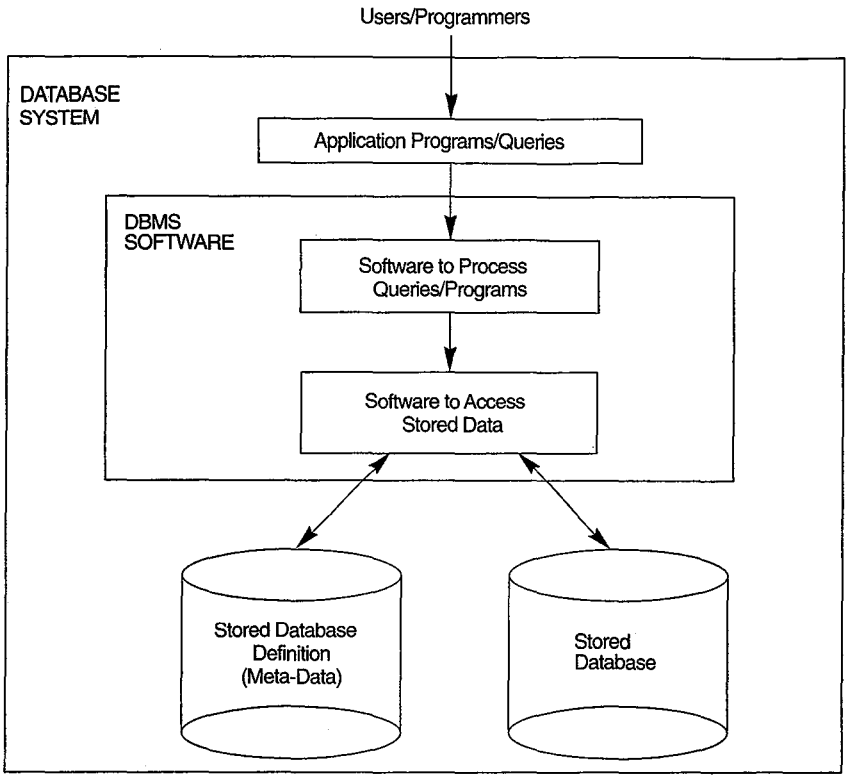


Figure 1.1 A simplified database system environment, illustrating the concepts and terminology discussed in Section 1.1.

the grades that students receive in the various sections they have completed; and the PREREQUISITE file stores the prerequisites of each course.

To *define* this database, we must specify the structure of the records of each file by specifying the different types of **data elements** to be stored in each record. In Figure 1.2, each STUDENT record includes data to represent the student's Name, StudentNumber, Class (freshman or 1, sophomore or 2, . . .), and Major (MATH, computer science or CS, . . .); each COURSE record includes data to represent the CourseName, CourseNumber, CreditHours, and Department (the department that offers the course); and so on. We must also specify a **data type** for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, StudentNumber of STUDENT is an integer, and Grade of GRADE_REPORT is a single character from the set {A, B, C, D, F, I}. We may also use a coding scheme to represent a data item. For example, in Figure 1.2 we represent the Class of a STUDENT as 1 for freshman, 2 for sophomore, 3 for junior, 4 for senior, and 5 for graduate student.

Petitioner Uber Ex-1052, 0031

STUDENT	Name	StudentNumber	Class	Major
	Smith	17	1	CS
	Brown	8	2	CS

COURSE	CourseName	CourseNumber	CreditHours	Department
	Intro to Computer Science	CS1310	4	CS
	Data Structures	CS3320	4	CS
	Discrete Mathematics	MATH2410	3	MATH
	Database	CS3380	3	CS

SECTION	SectionIdentifier	CourseNumber	Semester	Year	Instructor
	85	MATH2410	Fall	98	King
	92	CS1310	Fall	98	Anderson
	102	CS3320	Spring	99	Knuth
	112	MATH2410	Fall	99	Chang
	119	CS1310	Fall	99	Anderson
	135	CS3380	Fall	99	Stone

GRADE_REPORT	StudentNumber	SectionIdentifier	Grade
	17	112	B
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

PREREQUISITE	CourseNumber	PrerequisiteNumber
	CS3380	CS3320
	CS3380	MATH2410
	CS3320	CS1310

Figure 1.2 An example of a database that stores student records and their grades.

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

Petitioner Uber Ex-1052, 0032

large databases include many types of records and have *many relationships* among the records.

Database *manipulation* involves querying and updating. Examples of queries are "retrieve the transcript—a list of all courses and grades—of Smith"; "list the names of students who took the section of the Database course offered in fall 1999 and their grades in that section"; and "what are the prerequisites of the Database course?" Examples of updates are "change the class of Smith to Sophomore"; "create a new section for the Database course for this semester"; and "enter a grade of A for Smith in the Database section of last semester." These informal queries and updates must be specified precisely in the database system language before they can be processed.

1.3 Characteristics of the Database Approach

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

In the database approach, a single repository of data is maintained that is defined once and then is accessed by various users. The main characteristics of the database approach versus the file-processing approach are the following.

1.3.1 Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the system **catalog,** which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data,** and it describes the structure of the primary database (Figure 1.1).

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

Petitioner Uber Ex-1052, 0033

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only *one specific database*, whose structure is declared in the application programs. For example, a PASCAL program may have record structures declared in it; a C++ program may have "struct" or "class" declarations; and a COBOL program has Data Division statements to define its files. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and then using these definitions.

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

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the access programs, so any changes to the structure of a file may require *changing all programs* that access this file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence.** For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure 1.3. If we want to add another piece of data to each STUDENT record, say the Birthdate, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we just need to change the description of STUDENT records in the catalog to reflect the inclusion of the new data item Birthdate; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

In object-oriented and object-relational databases (see Part III), users can define operations on data as part of the database definitions. An **operation** (also called a *function*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation*

Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
StudentNumber	31	4
Class	35	4
Major	39	4

Figure 1.3 Internal storage format for a STUDENT record.

Petitioner Uber Ex-1052, 0034

(or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence.**

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

For example, consider again Figure 1.2. The internal implementation of a file may be defined by its record length—the number of characters (bytes) in each record—and each data item may be specified by its starting byte within a record and its length in bytes. The STUDENT record would thus be represented as shown in Figure 1.3. But a typical database user is not concerned with the location of each data item within a record or its length; rather the concern is that, when a reference is made to Name of STUDENT, the correct value is returned. A conceptual representation of the STUDENT records is shown in Figure 1.2. Many other details of file-storage organization—such as the access paths specified on a file—can be hidden from database users by the DBMS; we will discuss storage details in Chapters 5 and 6.

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

With the recent trend toward object-oriented and object-relational databases, abstraction is carried one level further to include not only the data structure but also the operations on the data. These operations provide an abstraction of miniworld activities commonly understood by the users. For example, an operation CALCULATE_GPA can be applied to a student object to calculate the grade point average. Such operations can be invoked by the user queries or programs without the user knowing the details of how they are internally implemented. In that sense, an abstraction of the miniworld activity is made available to the user as an **abstract operation.**

1.3.3 Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of applications must provide facilities for defining multiple

Petitioner Uber Ex-1052, 0035

(a)

TRANSCRIPT	StudentName	Student Transcript				
		CourseNumber	Grade	Semester	Year	SectionId
	Smith	CS1310	C	Fall	99	119
		MATH2410	B	Fall	99	112
	Brown	MATH2410	A	Fall	98	85
		CS1310	A	Fall	98	92
		CS3320	B	Spring	99	102
		CS3380	A	Fall	99	135

(b)

PREREQUISITES	CourseName	CourseNumber	Prerequisites
	Database	CS3380	CS3320
			MATH2410
	Data Structures	CS3320	CS1310

Figure 1.4 Two views derived from the example database shown in Figure 1.2. (a) The student transcript view. (b) The course prerequisite view.

views. For example, one user of the database of Figure 1.2 may be interested only in the transcript of each student; the view for this user is shown in Figure 1.4(a). A second user, who is interested only in checking that students have taken all the prerequisites of each course they register for, may require the view shown in Figure 1.4(b).

1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger. These types of applications are generally called **on-line transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly.

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

Petitioner Uber Ex-1052, 0036

1.4 Actors on the Scene

For a small personal database, such as the list of addresses discussed in Section 1.1, one person typically defines, constructs, and manipulates the database. However, many persons are involved in the design, use, and maintenance of a large database with a few hundred users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the "actors on the scene." In Section 1.5 we consider people who may be called "workers behind the scene"—those who work to maintain the database system environment, but who are not actively interested in the database itself.

1.4.1 Database Administrators

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

1.4.2 Database Designers

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

1.4.3 End Users

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

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.

Petitioner Uber Ex-1052, 0037

- **Naive or parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions**—that have been carefully programmed and tested. The tasks that such users perform are varied:

  Bank tellers check account balances and post withdrawals and deposits.

  Reservation clerks for airlines, hotels, and car rental companies check availability for a given request and make reservations.

  Clerks at receiving stations for courier mail enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.

- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.
- **Stand-alone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu- or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

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

1.4.4 System Analysts and Application Programmers (Software Engineers)

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

1.5 Workers behind the Scene

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

- **DBMS system designers and implementers** are persons who design and implement the DBMS modules and interfaces as a software package. A DBMS is a complex software system that consists of many components or **modules,** including modules for

Petitioner Uber Ex-1052, 0038

implementing the catalog, query language, interface processors, data access, concurrency control, recovery, and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.

- **Tool developers** include persons who design and implement **tools**—the software packages that facilitate database system design and use, and help improve performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation. In many cases, independent software vendors develop and market these tools.
- **Operators and maintenance personnel** are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

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

1.6 Advantages of Using a DBMS

In this section we discuss some of the advantages of using a DBMS and the capabilities that a good DBMS should possess. The DBA must utilize these capabilities to accomplish a variety of objectives related to the design, administration, and use of a large multiuser database.

1.6.1 Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example of Section 1.2; here, two groups of users might be the course registration personnel and the accounting office. In the traditional approach, each group independently keeps files on students. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Much of the data is stored twice: once in the files of each user group. Additional user groups may further duplicate some or all of the same data in their own files.

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

Petitioner Uber Ex-1052, 0039

(a)

GRADE_REPORT	StudentNumber	StudentName	SectionIdentifier	CourseNumber	Grade
	17	Smith	112	MATH2410	B
	17	Smith	119	CS1310	C
	8	Brown	85	MATH2410	A
	8	Brown	92	CS1310	A
	8	Brown	102	CS3320	B
	8	Brown	135	CS3380	A

(b)

GRADE_REPORT	StudentNumber	StudentName	SectionIdentifier	CourseNumber	Grade
	17	Brown	112	MATH2410	B

Figure 1.5 The redundant storage of data items. (a) *Controlled redundancy:* Including StudentName and CourseNumber in the GRADE_REPORT file. (b) *Uncontrolled redundancy:* A GRADE_REPORT record that is inconsistent with the STUDENT records in Figure 1.2, because the Name of student number 17 is Smith, not Brown.

enter a student's birthdate erroneously as JAN-19-1974, whereas the other user groups may enter the correct value of JAN-29-1974.

In the database approach, the views of different user groups are integrated during database design. For consistency, we should have a database design that stores each logical data item—such as a student's name or birth date—in *only one place* in the database. This does not permit inconsistency, and it saves storage space. However, in some cases, **controlled redundancy** may be useful for improving the performance of queries. For example, we may store StudentName and CourseNumber redundantly in a GRADE_REPORT file (Figure 1.5a), because, whenever we retrieve a GRADE_REPORT record, we want to retrieve the student name and course number along with the grade, student number, and section identifier. By placing all the data together, we do not have to search multiple files to collect this data. In such cases, the DBMS should have the capability to *control* this redundancy so as to prohibit inconsistencies among the files. This may be done by automatically checking that the StudentName-StudentNumber values in any GRADE_REPORT record in Figure 1.5(a) match one of the Name-StudentNumber values of a STUDENT record (Figure 1.2). Similarly, the SectionIdentifier-CourseNumber values in GRADE_REPORT can be checked against SECTION records. Such checks can be specified to the DBMS during database design and automatically enforced by the DBMS whenever the GRADE_REPORT file is updated. Figure 1.5(b) shows a GRADE_REPORT record that is inconsistent with the STUDENT file of Figure 1.2, which may be entered erroneously if the redundancy is *not controlled*.

1.6.2 Restricting Unauthorized Access

When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to

Petitioner Uber Ex-1052, 0040

retrieve and to update. Hence, the type of access operation—retrieval or update—must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS should provide a **security and authorization subsystem,** which the DBA uses to create accounts and to specify account restrictions. The DBMS should then enforce these restrictions automatically. Notice that we can apply similar controls to the DBMS software. For example, only the DBA's staff may be allowed to use certain **privileged software,** such as the software for creating new accounts. Similarly, parametric users may be allowed to access the database only through the canned transactions developed for their use.

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

Databases can be used to provide **persistent storage** for program objects and data structures. This is one of the main reasons for the emergence of the **object-oriented database systems.** Programming languages typically have complex data structures, such as record types in PASCAL or class definitions in C++. The values of program variables are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable structure. Object-oriented database systems are compatible with programming languages such as C++ and JAVA, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS, such as ObjectStore or O2 (now called Ardent, see Chapter 12). Such an object is said to be **persistent,** since it survives the termination of program execution and can later be directly retrieved by another C++ program.

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

1.6.4 Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems.** For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as **rules,** which when compiled and maintained by the DBMS can determine all students on probation. In a traditional DBMS, an explicit *procedural program code* would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. More

Petitioner Uber Ex-1052, 0041

powerful functionality is provided by **active database systems,** which provide active rules that can automatically initiate actions.

1.6.5 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users; programming language interfaces for application programmers; forms and command codes for parametric users; and menu-driven interfaces and natural language interfaces for stand-alone users. Both forms-style interfaces and menu-driven interfaces are commonly known as **graphical user interfaces (GUIs).** Many specialized languages and environments exist for specifying GUIs. Capabilities for providing World Wide Web access to a database—or web-enabling a database—are also becoming increasingly common.

1.6.6 Representing Complex Relationships Among Data

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

1.6.7 Enforcing Integrity Constraints

Most database applications have certain **integrity constraints** that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item. For example, in Figure 1.2, we may specify that the value of the Class data item within each student record must be an integer between 1 and 5 and that the value of Name must be a string of no more than 30 alphabetic characters. A more complex type of constraint that occurs frequently involves specifying that a record in one file must be related to records in other files. For example, in Figure 1.2, we can specify that "every section record must be related to a course record." Another type of constraint specifies uniqueness on data item values, such as "every course record must have a unique value for CourseNumber." These constraints are derived from the meaning or **semantics** of the data and of the miniworld it represents. It is the database designers' responsibility to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry.

A data item may be entered erroneously and still satisfy the specified integrity constraints. For example, if a student receives a grade of A but a grade of C is entered in the database, the DBMS *cannot* discover this error automatically, because C is a valid value for

Petitioner Uber Ex-1052, 0042

the Grade data type. Such data entry errors can only be discovered manually (when the student receives the grade and complains) and corrected later by updating the database. However, a grade of Z can be rejected automatically by the DBMS, because Z is not a valid value for the Grade data type.

1.6.8 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update program, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the program started executing. Alternatively, the recovery subsystem could ensure that the program is resumed from the point at which it was interrupted so that its full effect is recorded in the database.

1.7 Implications of the Database Approach

In addition to the issues discussed in the previous section, there are other implications of using the database approach that can benefit most organizations.

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

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

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

Availability of Up-to-Date Information. A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immedi-

Petitioner Uber Ex-1052, 0043

ately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

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

1.8 When Not to Use a DBMS

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

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

Additional problems may arise if the database designers and DBA do not properly design the database or if the database systems applications are not implemented properly. Hence, it may be more desirable to use regular files under the following circumstances:

- The database and applications are simple, well defined, and not expected to change.
- There are stringent real-time requirements for some programs that may not be met because of DBMS overhead.
- Multiple-user access to data is not required.

1.9 Summary

In this chapter we defined a database as a collection of related data, where *data* means recorded facts. A typical database represents some aspect of the real world and is used for specific purposes by one or more groups of users. A DBMS is a generalized software package for implementing and maintaining a computerized database. The database and software together form a database system. We identified several characteristics that distinguish the database approach from traditional file-processing applications:

- Existence of a catalog.
- Program-data independence and program-operation independence.
- Data abstraction.
- Support of multiple user views.
- Sharing of data among multiple transactions.

Petitioner Uber Ex-1052, 0044

We then discussed the main categories of database users, or the "actors on the scene":

- Administrators.
- Designers.
- End users.
- System analysts and application programmers.

We noted that, in addition to database users, there are several categories of support personnel, or "workers behind the scene," in a database environment:

- DBMS system designers and implementers.
- Tool developers.
- Operators and maintenance personnel.

Then we presented a list of capabilities that should be provided by the DBMS software to the DBA, database designers, and users to help them design, administer, and use a database:

- Controlling redundancy.
- Restricting unauthorized access.
- Providing persistent storage for program objects and data structures.
- Permitting inferencing and actions by using rules.
- Providing multiple user interfaces.
- Representing complex relationships among data.
- Enforcing integrity constraints.
- Providing backup and recovery.

We listed some additional advantages of the database approach over traditional file-processing systems:

- Potential for enforcing standards.
- Reduced application development time.
- Flexibility.
- Availability of up-to-date information to all users.
- Economies of scale.

Finally, we discussed the overhead costs of using a DBMS and discussed some situations in which it may not be advantageous to use a DBMS.

Review Questions

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

Petitioner Uber Ex-1052, 0045

1.2. What three main types of actions involve databases? Briefly discuss each.
1.3. Discuss the main characteristics of the database approach and how it differs from traditional file systems.
1.4. What are the responsibilities of the DBA and the database designers?
1.5. What are the different types of database end users? Discuss the main activities of each.
1.6. Discuss the capabilities that should be provided by a DBMS.

Exercises

1.7. Identify some informal queries and update operations that you would expect to apply to the database shown in Figure 1.2.
1.8. What is the difference between controlled and uncontrolled redundancy? Illustrate with examples.
1.9. Name all the relationships among the records of the database shown in Figure 1.2.
1.10. Give some additional views that may be needed by other user groups for the database shown in Figure 1.2.
1.11. Cite some examples of integrity constraints that you think should hold on the database shown in Figure 1.2.

Selected Bibliography

The October 1991 issue of *Communications of the* ACM and Kim (1995) includes several articles describing "next-generation" DBMSs; many of the database features discussed in this issue are now commercially available. The March 1976 issue of ACM *Computing Surveys* offers an early introduction to database systems and may provide a historical perspective for the interested reader.

Petitioner Uber Ex-1052, 0046

CHAPTER 2 Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package is one tightly integrated system, to the modern DBMS packages that are modular in design, with a client-server system architecture. This evolution mirrors the trends in computing, where the large centralized mainframe computers are being replaced by hundreds of distributed workstations and personal computers connected via communications networks. In a basic client-server architecture, the system functionality is distributed between two types of modules. A **client module** is typically designed so that it will run on a user workstation or personal computer. Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as forms or menu-based GUIs (graphical user interfaces). The other kind of module, called a **server module,** typically handles data storage, access, search, and other functions.

We will discuss client-server architectures in Chapters 17 and 24. First, we must study more basic concepts that will give us a better understanding of the modern database architectures when they are presented later in this book. In this chapter we thus present the terminology and basic concepts that will be used throughout the book. We start, in Section 2.1, by discussing data models and defining the concepts of schemas and instances, which are fundamental to the study of database systems. We then discuss the three-schema DBMS architecture and data independence in Section 2.2; this provides a user's perspective on what a DBMS is supposed to do. In Section 2.3, we describe the types

Petitioner Uber Ex-1052, 0047

of interfaces and languages that are typically provided by a DBMS. Section 2.4 discusses the database system software environment, and Section 2.5 presents a classification of the types of DBMS packages. Section 2.6 summarizes the chapter.

The material in Sections 2.4 and 2.5 provides more detailed concepts that may be looked upon as a supplement to the basic introductory material.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction by hiding details of data storage that are not needed by most database users. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.[1] By *structure of a database* we mean the data types, relationships, and constraints that should hold on the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the **dynamic aspect** or **behavior** of a database application. This allows the database designer to specify a set of valid **user-defined operations** that are allowed on the database objects.[2] An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the *basic data model operations*. Concepts to specify behavior are fundamental to object-oriented data models (see Chapters 11 and 12) but are also being incorporated in more traditional data models by extending these models. For example, object-relational models (see Chapter 13) extend the traditional relational model to include such concepts, among others.

2.1.1 Categories of Data Models

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

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

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

Petitioner Uber Ex-1052, 0048

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project, that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project. In Chapter 3, we will present the Entity-Relationship model—a popular high-level conceptual data model. Chapter 4 describes additional data modeling concepts, such as generalization, specialization, and categories.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used **relational data model,** as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Part II of this book is devoted to the relational data model, its operations and languages, and also includes an overview of two relational systems.[3] The SQL standard for relational databases is described in Chapter 8. Representational data models represent data by using record structures and hence are sometimes called **record-based data models.**

We can regard **object data models** as a new family of higher-level implementation data models that are closer to conceptual data models. We describe the general characteristics of object databases, together with an overview of two object DBMSs, in Part III of this book. The ODMG proposed standard for object databases is described in Chapter 12. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored in the computer by representing information such as record formats, record orderings, and access paths. An **access path** is a structure that makes the search for particular database records efficient. We discuss physical storage techniques and access structures in Chapters 5 and 6.

2.1.2 Schemas, Instances, and Database State

In any data model it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the **database schema,** which is specified during database design and is not expected to change frequently.[4] Most data models have certain conventions for displaying the schemas as diagrams.[5] A displayed schema is called a **schema diagram.** Figure 2.1 shows a schema diagram for the database shown in Figure 1.2; the diagram displays the structure of each record type but not the actual instances of records. We call each object in the schema—such as STUDENT or COURSE—a **schema construct.**

3. A summary of the network and hierarchical data models is included in Appendices C and D. The full chapters from the second edition of this book are accessible from the Internet Web site.

4. Schema changes are usually needed as the requirements of the database applications change. Newer database systems include operations for allowing schema changes, although the schema change process is more involved than simple database updates.

5. It is customary in database parlance to use *schemas* as plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is sometimes used for schema.

Petitioner Uber Ex-1052, 0049

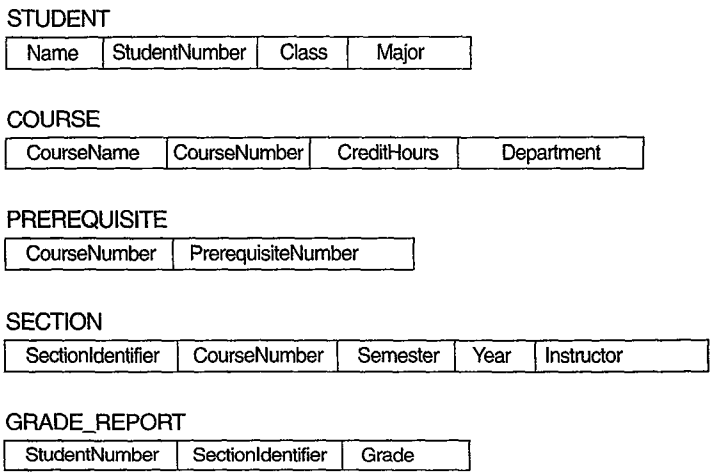


Figure 2.1 Schema diagram for the database of Figure 1.2.

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

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

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.[6] The DBMS is partly responsible for ensuring that *every* state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a

6. The current state is also called the *current snapshot* of the database.

Petitioner Uber Ex-1052, 0050

correct schema to the DBMS is extremely important, and the schema must be designed with the utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension,** and a database state an **extension** of the schema.

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

2.2 DBMS Architecture and Data Independence

Three important characteristics of the database approach, listed in Section 1.3, are (1) insulation of programs and data (program-data and program-operation independence); (2) support of multiple user views; and (3) use of a catalog to store the database description (schema). In this section we specify an architecture for database systems, called the **three-schema architecture,**[7] which was proposed to help achieve and visualize these characteristics. We then discuss the concept of data independence.

2.2.1 The Three-Schema Architecture

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

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

7. This is also known as the ANSI/SPARC architecture, after the committee that proposed it (Tsichritzis and Klug 1978).

Petitioner Uber Ex-1052, 0051

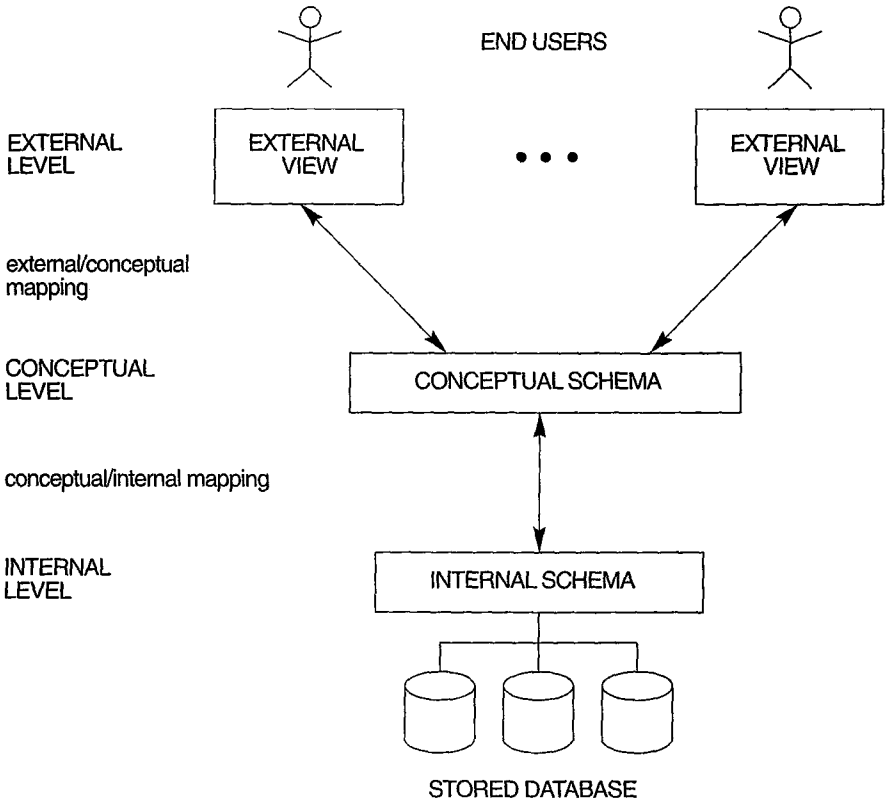


Figure 2.2 Illustrating the three-schema architecture.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

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

Petitioner Uber Ex-1052, 0052

external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

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

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.4(a) should not be affected by changing the GRADE_REPORT file shown in Figure 1.2 into the one shown in Figure 1.5(a). Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval of SECTION records (Figure 1.2) by Semester and Year should not require a query such as "list all sections offered in fall 1998" to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

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

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

Petitioner Uber Ex-1052, 0053

2.3 Database Languages and Interfaces

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

2.3.1 DBMS Languages

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

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language (SDL)**, is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. For a true three-schema architecture, we would need a third language, the **view definition language (VDL)**, to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

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

In current DBMSs, the preceding types of languages are usually *not considered distinct languages*; rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, and it is usually utilized by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language (see Chapter 8), which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification and schema evolution. The SDL was a component in earlier versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A **high-level** or **nonprocedural DML** can be used on its own to specify complex database operations in a concise manner. Many DBMSs allow high-level DML statements either to be entered interactively from a terminal (or monitor) or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a pre-compiler and processed by the DBMS. A **low-level** or **procedural DML** *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately.

Petitioner Uber Ex-1052, 0054

Hence, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time DMLs** because of this property. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement and are hence called **set-at-a-time** or **set-oriented DMLs.** A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; hence, such languages are also called **declarative.**

Whenever DML commands, whether high-level or low-level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage.**[8] On the other hand, a high-level DML used in a stand-alone interactive manner is called a **query language.** In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.[9]

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

2.3.2 DBMS Interfaces

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

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

Forms-Based Interfaces. A forms-based interface displays a **form** to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages,** special languages that help programmers specify such forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

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

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

Petitioner Uber Ex-1052, 0055

Graphical User Interfaces. A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device,** such as a mouse, to pick certain parts of the displayed schema diagram.

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

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for a known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

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

2.4 The Database System Environment

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

2.4.1 DBMS Component Modules

Figure 2.3 illustrates, in a simplified form, the typical DBMS components. The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system (OS),** which schedules disk input/output. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog. The dotted lines and circles marked A, B, C, D, and E in Figure 2.3 illustrate accesses that are under the control of this stored data manager. The stored data manager may use basic OS services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory. Once the data is in main memory buffers, it can be processed by other DBMS modules, as well as by application programs.

Petitioner Uber Ex-1052, 0056

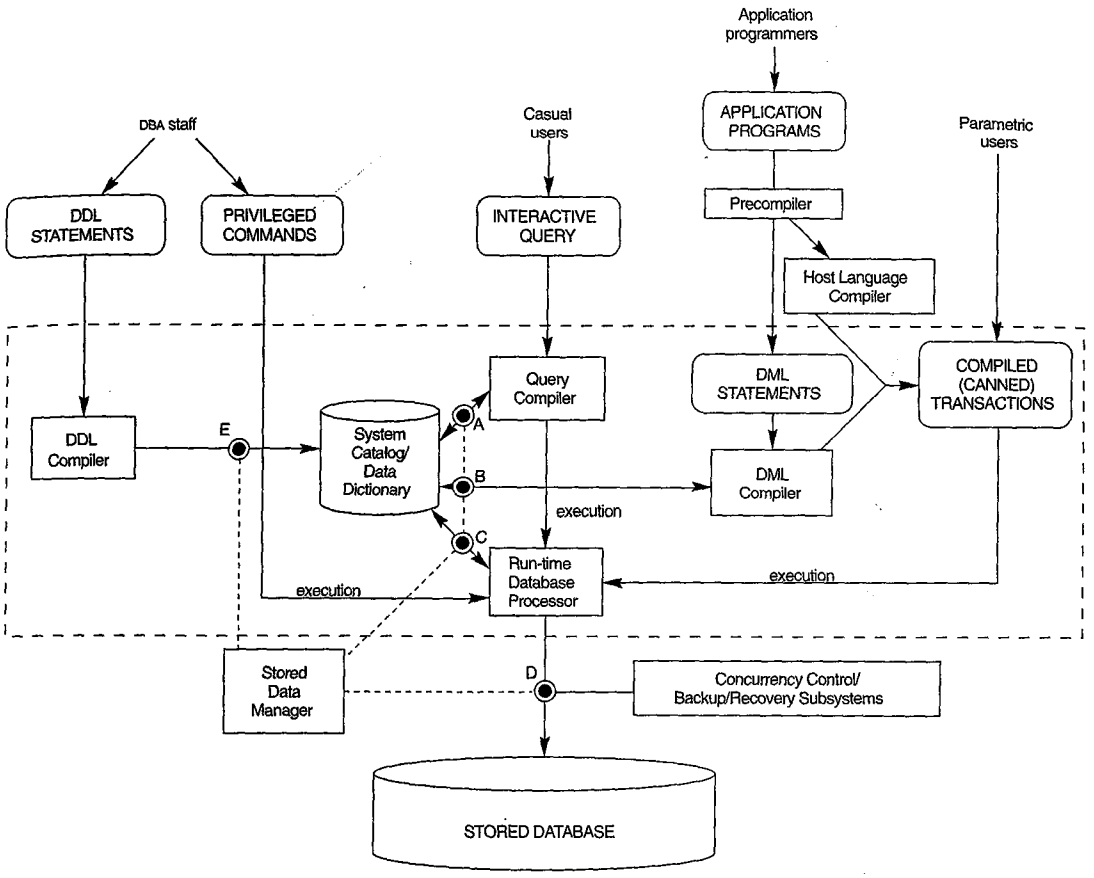


Figure 2.3 Typical component modules of a DBMS. Dotted lines show accesses that are under the control of the stored data manager.

The **DDL compiler** processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file, mapping information among schemas, and constraints, in addition to many other types of information that are needed by the DBMS modules. DBMS software modules then look up the catalog information as needed.

The **run-time database processor** handles database accesses at run time; it receives retrieval or update operations and carries them out on the database. Access to disk goes through the stored data manager. The **query compiler** handles high-level queries that are entered interactively. It parses, analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

Petitioner Uber Ex-1052, 0057

The **pre-compiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the DML **compiler** for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.

Figure 2.3 is not meant to describe a specific DBMS; rather it illustrates typical DBMS modules. The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. The DBMS also interfaces with compilers for general-purpose host programming languages. User-friendly interfaces to the DBMS can be provided to help any of the user types shown in Figure 2.3 to specify their requests.

2.4.2 Database System Utilities

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

1. *Loading*: A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called **conversion tools.**
2. *Backup:* A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The backup copy can be used to restore the database in case of catastrophic failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex but it saves space.
3. *File reorganization:* This utility can be used to reorganize a database file into a different file organization to improve performance.
4. *Performance monitoring:* Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files to improve performance.

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

Petitioner Uber Ex-1052, 0058

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and DBAs. CASE **tools**[10] are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**) **system.** In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository.** This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

Application development environments, such as the PowerBuilder system, are becoming quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

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

2.5 Classification of Database Management Systems

Several criteria are normally used to classify DBMSs. The first is the **data model** on which the DBMS is based. The two types of data models used in many current commercial DBMSs are the **relational data model** and the **object data model.** Many legacy applications still run on database systems based on the **hierarchical** and **network data models.** The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs that are being called **object-relational DBMSs.** We can hence categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

10. Although CASE stands for Computer Aided Software Engineering, many CASE tools are used primarily for database design.

Petitioner Uber Ex-1052, 0059

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

A third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A **distributed DBMS (DDBMS)** can have the actual database and DBMS software distributed over many sites, connected by a computer network. **Homogeneous DDBMSs** use the same DBMS software at multiple sites. A recent trend is to develop software to access several autonomous preexisting databases stored under **heterogeneous DBMSs.** This leads to a **federated** DBMS (or **multidatabase system**), where the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use a client-server architecture.

A fourth criterion is the **cost** of the DBMS. The majority of DBMS packages cost between $10,000 and $100,000. Single-user low-end systems that work with microcomputers cost between $100 and $3000. At the other end, a few elaborate packages cost more than $100,000.

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

Let us briefly elaborate on the main criterion for classifying DBMSs: the data model. The basic relational data model represents a database as a collection of tables, where each table can be stored as a separate file. The database in Figure 1.2 is shown in a manner very similar to a relational representation. Most relational databases use the high-level query language called SQL and support a limited form of user views. We discuss the relational model, its languages and operations, and two sample commercial systems in Chapters 7 through 10.

The object data model defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class,** and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of each class are specified in terms of predefined procedures called **methods.** Relational DBMSs have been extending their models to incorporate object database concepts and other capabilities; these systems are referred to as **object-relational** or **extended-relational systems.** We discuss object databases and extended-relational systems in Chapters 11 through 13.

The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type.** Figure 2.4 shows a network schema diagram for the database of Figure 1.2, where record types are shown as rectangles and set types are shown

Petitioner Uber Ex-1052, 0060

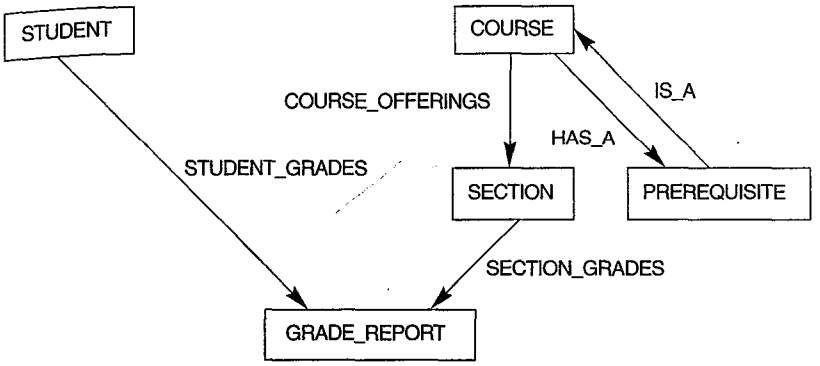


Figure 2.4 The schema of Figure 2.1 in the notation of the network data model.

as labeled directed arrows. The network model, also known as the CODASYL DBTG model,[11] has an associated record-at-a-time language that must be embedded in a host programming language. The **hierarchical model** represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model, although most hierarchical DBMSs have record-at-a-time languages. We give a brief overview of the network and hierarchical models in Appendices C and D.[12]

2.6 Summary

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

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

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

- An internal schema describes the physical storage structure of the database.

11. CODASYL DBTG stands for Computer Data Systems Language Data Base Task Group, which is the committee that specified the network model and its language.

12. The full chapters on the network and hierarchical models from the second edition of this book are available over the Internet from the Web site.

Petitioner Uber Ex-1052, 0061

- A conceptual schema is a high-level description of the whole database.
- External schemas describe the views of different user groups.

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

We then discussed the main types of languages and interfaces that DBMSs support. A data definition language (DDL) is used to define the database conceptual schema. In most DBMSs, the DDL also defines user views and, sometimes, storage structures; in other DBMSs, separate languages (VDL, SDL) may exist for specifying views and storage structures. The DBMS compiles all schema definitions and stores their descriptions in the DBMS catalog. A data manipulation language (DML) is used for specifying database retrievals and updates. DMLs can be high-level (set-oriented, nonprocedural) or low-level (record-oriented, procedural). A high-level DML can be embedded in a host programming language, or it can be used as a stand-alone language; in the latter case it is often called a query language.

We discussed different types of interfaces provided by DBMSs, and the types of DBMS users with which each interface is associated. We then discussed the database system environment, typical DBMS software modules, and DBMS utilities for helping users and the DBA perform their tasks.

In the final section, we classified DBMSs according to several criteria: data model, number of users, number of sites, cost, types of access paths, and generality. The main classification of DBMSs is based on the data model. We briefly discussed the main data models used in current commercial DBMSs.

Review Questions

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

2.2. Discuss the main categories of data models.

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

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

2.5. What is the difference between logical data independence and physical data independence?

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

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

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

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

Petitioner Uber Ex-1052, 0062

Exercises

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

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

Selected Bibliography

Many database textbooks, including Date (1995), Silberschatz et al. (1998), Ramakrishnan (1997), Ullman (1988, 1989), and Abiteboul et al. (1995), provide a discussion of the various database concepts presented here. Tsichritzis and Lochovsky (1982) is an early textbook on data models. Tsichritzis and Klug (1978) and Jardine (1977) present the three-schema architecture, which was first suggested in the DBTG CODASYL report (1971) and later in an American National Standards Institute (ANSI) report (1975). An in-depth analysis of the relational data model and some of its possible extensions is given in Codd (1992). The proposed standard for object-oriented databases is described in Cattell (1997).

An example of database utilities is the ETI Extract Toolkit (www.eti.com) and the database administration tool DB Artisan from Embarcadero Technologies (www.embarcadero.com).

Petitioner Uber Ex-1052, 0063

CHAPTER 3 Data Modeling Using the Entity-Relationship Model

Conceptual modeling is an important phase in designing a successful database application. Generally, the term **database application** refers to a particular database—for example, a BANK database that keeps track of customer accounts—and the associated *programs* that implement the database queries and updates—for example, programs that implement database updates corresponding to customers making deposits and withdrawals. These programs often provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus. Hence, part of the database application will require the design, implementation, and testing of these **application programs.** Traditionally, the design and testing of application programs has been considered to be more in the realm of the software engineering domain than in the database domain. However, it is becoming clearer that there is some commonality between database design methodologies and software engineering design methodologies. As database design methodologies attempt to include more of the concepts for specifying operations on database objects, and as software engineering methodologies specify in more detail the structure of the databases that software programs will use and access, it is certain that this commonality will increase. We will briefly discuss some of the concepts for specifying database operations in Chapter 4, and again when we discuss object databases in Part III of this book.

In this chapter, we will follow the traditional approach of concentrating on the database structures and constraints during database design. We will present the modeling concepts of the **Entity-Relationship (ER) model,** which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of

Petitioner Uber Ex-1052, 0064

database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications.

This chapter is organized as follows. In Section 3.1 we discuss the role of high-level conceptual data models in database design. We introduce the requirements for an example database application in Section 3.2 to illustrate the use of the ER model concepts. This example database is also used in subsequent chapters. In Section 3.3 we present the concepts of entities and attributes, and we gradually introduce the diagrammatic technique for displaying an ER schema. In Section 3.4, we introduce the concepts of binary relationships and their roles and structural constraints. Section 3.5 introduces weak entity types. Section 3.6 shows how a schema design is refined to include relationships. Section 3.7 reviews the notation for ER diagrams, summarizes the issues that arise in schema design, and discusses how to choose the names for database schema constructs. Section 3.8 summarizes the chapter.

The material in Sections 3.3 and 3.4 provides a somewhat detailed description, and some may be left out of an introductory course if desired. On the other hand, if more thorough coverage of data modeling concepts and conceptual database design is desired, the reader should continue on to the material in Chapter 4 after concluding Chapter 3. In Chapter 4, we describe extensions to the ER model that lead to the Enhanced-ER (EER) model, which includes concepts such as specialization, generalization, inheritance, and union types (categories). We also introduce object modeling and the Universal Modeling Language (UML) notation in Chapter 4, which has been proposed as a standard for object modeling.

3.1 Using High-Level Conceptual Data Models for Database Design

Figure 3.1 shows a simplified description of the database design process. The first step shown is **requirements collection and analysis.** During this step, the database designers interview prospective database users to understand and document their **data requirements.** The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, and they include both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques for specifying functional requirements. We will not discuss any of these techniques here because they are usually part of software engineering texts.

Once all the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design.** The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are

Petitioner Uber Ex-1052, 0065

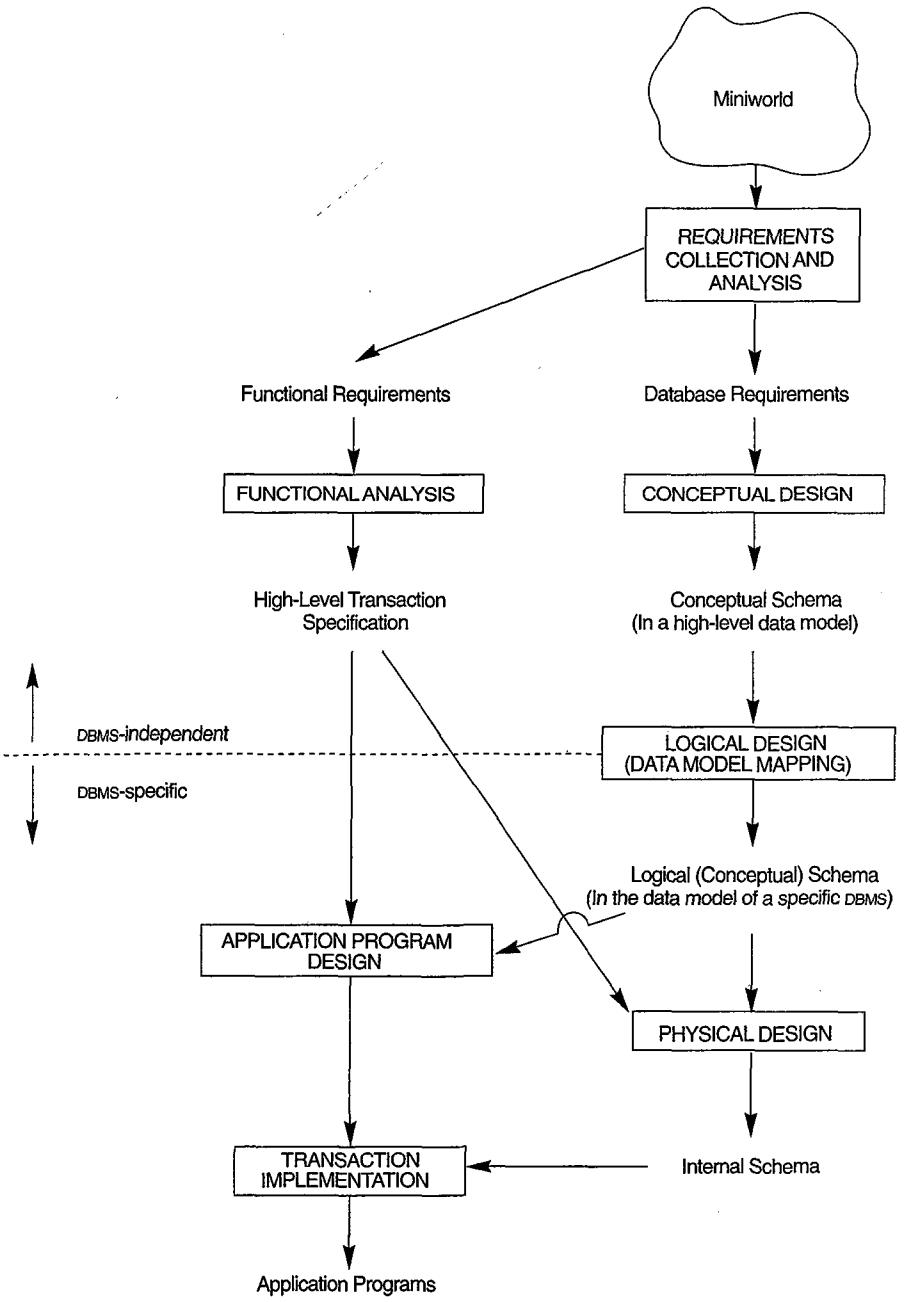


Figure 3.1 A simplified diagram to illustrate the main phases of database design.

Petitioner Uber Ex-1052, 0066

usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not include conflicts. This approach enables the database designers to concentrate on specifying the properties of the data, without being concerned with storage details. Consequently, it is easier for them to come up with a good conceptual database design.

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

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping,** and its result is a database schema in the implementation data model of the DBMS.

Finally, the last step is the **physical design** phase, during which the internal storage structures, access paths, and file organizations for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications. We will discuss the database design process in more detail, including an overview of physical database design, in Chapter 16.

We present only the ER model concepts for conceptual schema design in this chapter. The incorporation of user-defined operations is discussed in Chapter 4, when we introduce object modeling.

3.2 An Example Database Application

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

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

Petitioner Uber Ex-1052, 0067

3. We store each employee's name, social security number,[1] address, salary, sex, and birth date. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.
4. We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

Figure 3.2 shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams.** We describe the process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts in the following section.

3.3 Entity Types, Entity Sets, Attributes, and Keys

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

3.3.1 Entities and Attributes

Entities and Their Attributes. The basic object that the ER model represents is an **entity,** which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence—a particular person, car, house, or employee—or it may be an object with a conceptual existence—a company, a job, or a university course. Each entity has **attributes**—the particular properties that describe it. For example, an employee entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a **value** for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

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

Several types of attributes occur in the ER model: *simple* versus *composite; single-valued* versus *multivalued;* and *stored* versus *derived.* We first define these attribute types and illustrate their use via examples. We then introduce the concept of a *null value* for an attribute.

1. The social security number, or SSN, is a unique 9-digit identifier assigned to each individual in the United States to keep track of their employment, benefits, and taxes. Other countries may have similar identification schemes, such as personal identification card numbers.

Petitioner Uber Ex-1052, 0068

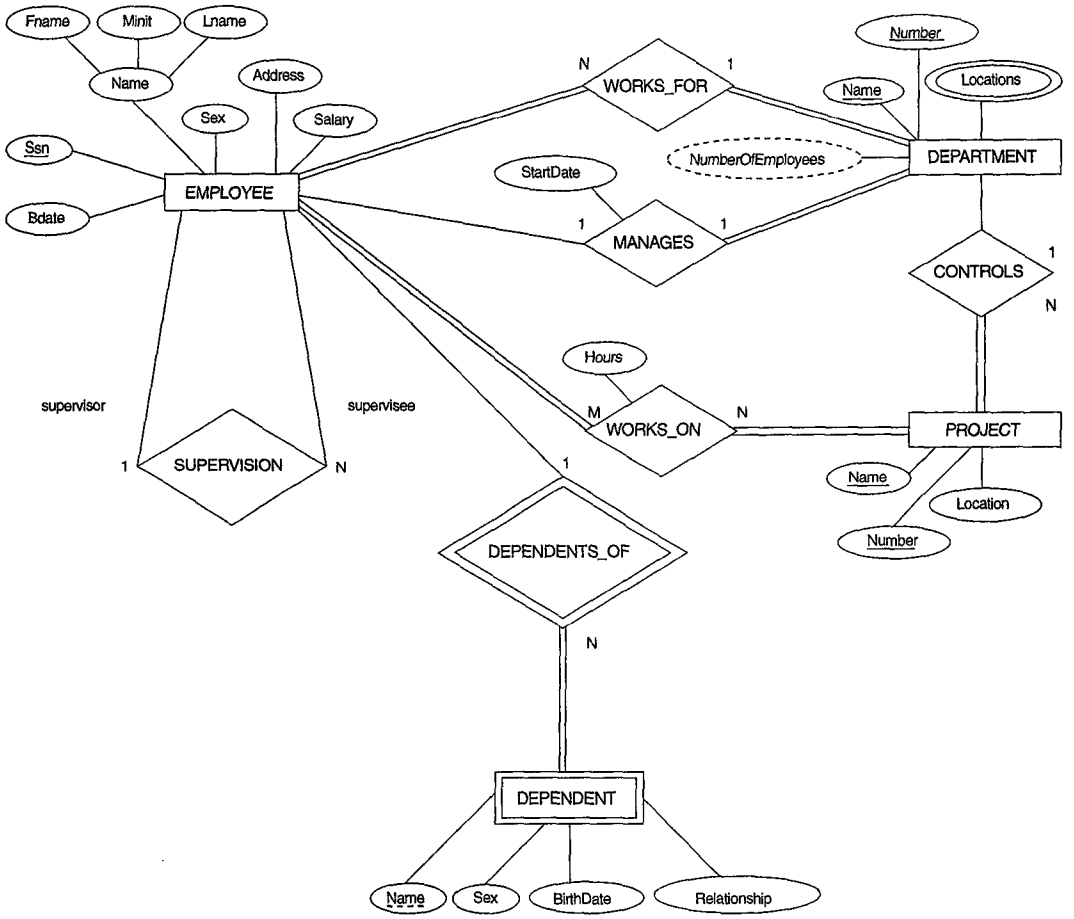


Figure 3.2 ER schema diagram for the COMPANY database.

Composite Versus Simple (Atomic) Attributes. **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the employee entity shown in Figure 3.3 can be subdivided into StreetAddress, City, State, and Zip,[2] with the values "2311 Kirby," "Houston," "Texas," and "77001." Attributes that are not divisible are called **simple** or **atomic attributes.** Composite attributes can form a hierarchy; for example, StreetAddress can be subdivided into three simple attributes, Number, Street, and ApartmentNumber, as shown in Figure 3.4. The value of a composite attribute is the concatenation of the values of its constituent simple attributes.

2. The zip code is the name used in the United States for a postal code.

Petitioner Uber Ex-1052, 0069

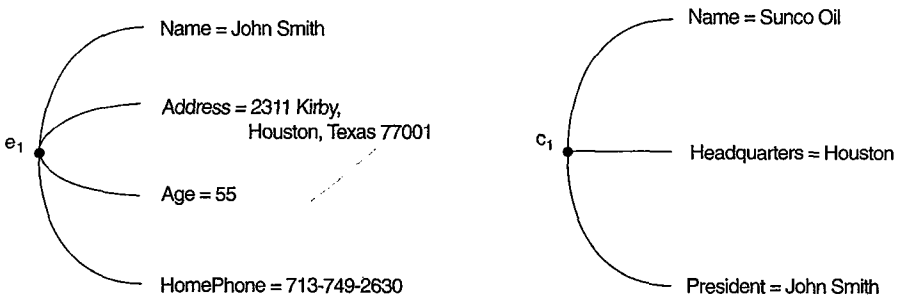


Figure 3.3 Two entities, an employee e_1 and a company c_1, and their attribute values.

Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the composite attribute is referenced only as a whole, there is no need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (Zip, Street, and so on), then the whole address is designated as a simple attribute.

Single-valued Versus Multivalued Attributes. Most attributes have a single value for a particular entity; such attributes are called **single-valued.** For example, Age is a single-valued attribute of person. In some cases an attribute can have a set of values for the same entity—for example, a Colors attribute for a car, or a CollegeDegrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two values for Colors. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; so different persons can have

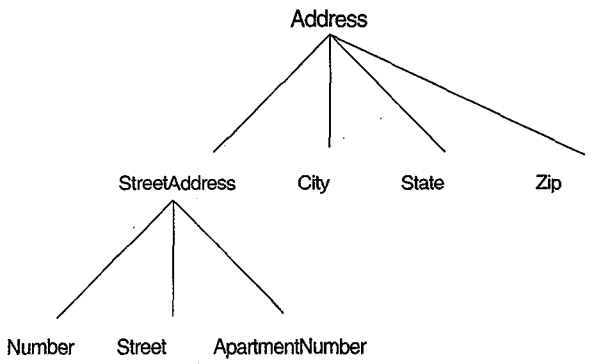


Figure 3.4 A hierarchy of composite attributes; the StreetAddress component of an Address is further composed of Number, Street, and ApartmentNumber.

Petitioner Uber Ex-1052, 0070

different *numbers of values* for the CollegeDegrees attribute. Such attributes are called **multivalued.** A multivalued attribute may have lower and upper bounds on the number of values allowed for each individual entity. For example, the Colors attribute of a car may have between one and three values, if we assume that a car can have at most three colors.

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

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

Complex Attributes. Notice that composite and multivalued attributes can be nested in an arbitrary way. We can represent arbitrary nesting by grouping components of a composite attribute between parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces {}. Such attributes are called **complex attributes.** For example, if a person can have more than one residence and each residence can have multiple phones, an attribute AddressPhone for a PERSON entity type can be specified as shown in Figure 3.5.

{AddressPhone({Phone(AreaCode, PhoneNumber)},
Address(StreetAddress(Number, Street, ApartmentNumber),City, State, Zip)) }

Figure 3.5 A complex attribute AddressPhone with multivalued and composite components.

Petitioner Uber Ex-1052, 0071

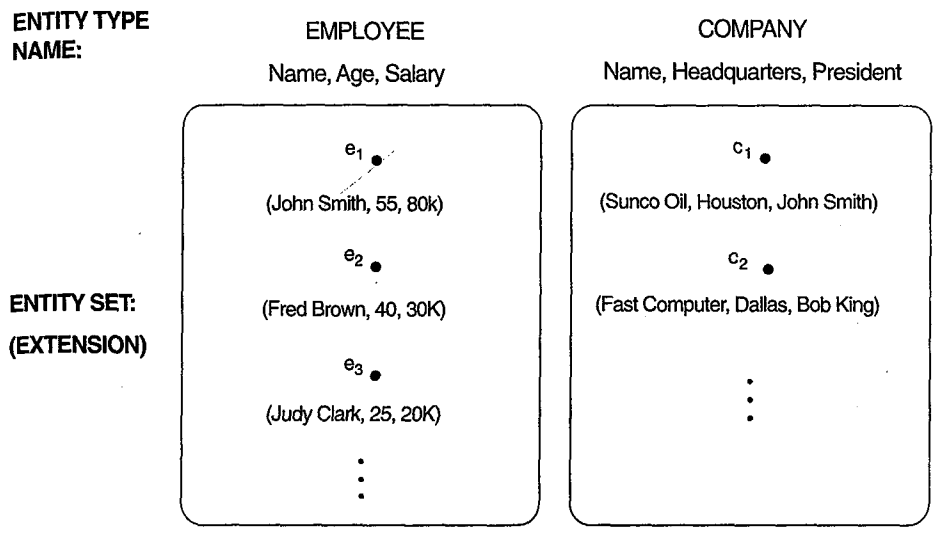


Figure 3.6 Two entity types named EMPLOYEE and COMPANY, and some of the member entities in the collection of entities (or entity set) of each type.

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

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

An entity type is represented in ER diagrams[3] (see Figure 3.2) as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

3. We are using a notation for ER diagrams that is close to the original proposed notation (Chen 1976). Unfortunately, many other notations are in use. We illustrate some of the other notations in Appendix A and later in this chapter.

Petitioner Uber Ex-1052, 0072

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

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has an attribute whose values are distinct for each individual entity in the collection. Such an attribute is called a **key attribute,** and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 3.6, because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is SocialSecurityNumber. Sometimes, several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, we can define a *composite attribute* that becomes a key attribute of the entity type. Notice that a composite key must be *minimal;* that is, all component attributes must be included in the composite attribute to have the uniqueness property.[4] In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure 3.2.

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

Some entity types have *more than one* key attribute. For example, each of the VehicleID and Registration attributes of the entity type CAR (Figure 3.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, RegistrationNumber and State, neither of which is a key on its own. An entity type may also have *no key,* in which case it is called a *weak entity type* (see Section 3.5).

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

Mathematically, an attribute A of entity type E whose value set is V can be defined as a **function** from E to the power set[5] $P(V)$ of V:

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

4. Superfluous attributes must not be included in a key; however, a **superkey** may include superfluous attributes, as we explain in Chapter 7.

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

Petitioner Uber Ex-1052, 0073

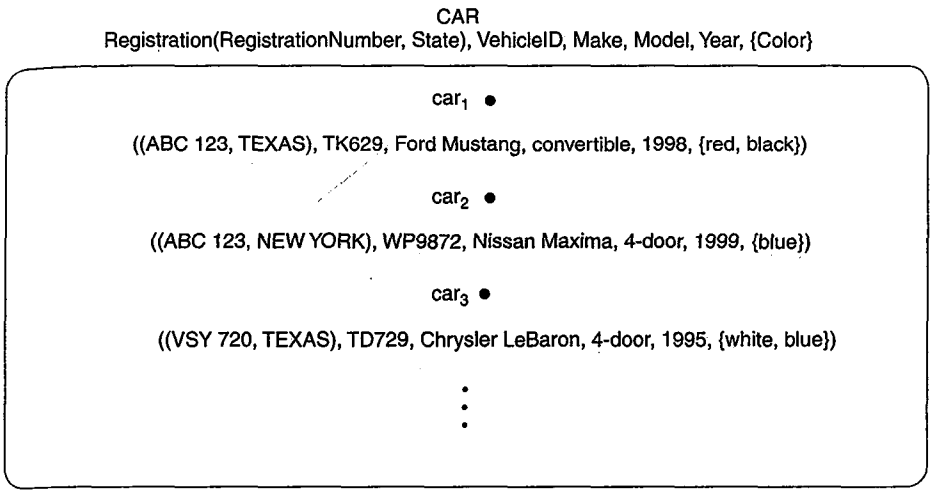


Figure 3.7 The CAR entity type, with two key attributes Registration and VehicleID. Multivalued attributes are shown between set braces {}. Components of a composite attribute are shown between parentheses ().

We refer to the value of attribute A for entity e as $A(e)$. The previous definition covers both single-valued and multivalued attributes, as well as nulls. A null value is represented by the empty set. For single-valued attributes, $A(e)$ is restricted to being a singleton for each entity e in E whereas there is no restriction on multivalued attributes.[6] For a composite attribute A, the value set V is the Cartesian product of $P(V_1)$, $P(V_2)$, . . ., $P(V_n)$, where $V_1, V_2, \ldots, V_n$ are the value sets of the simple component attributes that form A:

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

3.3.3 Initial Conceptual Design of the COMPANY Database

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

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

6. A **singleton** is a set with only one element (value).

Petitioner Uber Ex-1052, 0074

DEPARTMENT
Name, Number, {Locations}, Manager, ManagerStartDate

PROJECT
Name, Number, Location, ControllingDepartment

EMPLOYEE
Name (FName, MInit, LName), SSN, Sex, Address, Salary,
BirthDate, Department, Supervisor, {WorksOn (Project, Hours)}

DEPENDENT
Employee, DependentName, Sex, BirthDate, Relationship

Figure 3.8 Preliminary design of entity types for the COMPANY database whose requirements are described in Section 3.2.

2. An entity type PROJECT with attributes Name, Number, Location, and ControllingDepartment. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, SSN (for social security number), Sex, Address, Salary, BirthDate, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—FirstName, MiddleInitial, LastName—or of Address.
4. An entity type DEPENDENT with attributes Employee, DependentName, Sex, BirthDate, and Relationship (to the employee).

So far, we have not represented the fact that an employee can work on several projects, nor have we represented the number of hours per week an employee works on each project. This characteristic is listed as part of requirement 3 in Section 3.2, and it can be represented by a multivalued composite attribute of EMPLOYEE called WorksOn with simple components (Project, Hours). Alternatively, it can be represented as a multivalued composite attribute of PROJECT called Workers with simple components (Employee, Hours). We choose the first alternative in Figure 3.8, which shows each of the entity types described above. The Name attribute of EMPLOYEE is shown as a composite attribute, presumably after consultation with the users.

3.4 Relationships, Relationship Types, Roles, and Structural Constraints

In Figure 3.8 there are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department; the attribute ControllingDepartment of PROJECT refers to the department that controls the project; the attribute Supervisor of EMPLOYEE refers to another

Petitioner Uber Ex-1052, 0075

employee (the one who supervises this employee); the attribute Department of EMPLOYEE refers to the department for which the employee works; and so on. In the ER model, these references should not be represented as attributes but as **relationships,** which are discussed in this section. The COMPANY database schema will be refined in Section 3.6 to represent relationships explicitly. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

This section is organized as follows. Section 3.4.1 introduces the concepts of relationship types, sets, and instances. Section 3.4.2 defines the concepts of relationship degree, role names, and recursive relationships. Section 3.4.3 discusses structural constraints on relationships, such as cardinality ratios (1:1, 1:N, M:N) and existence dependencies. Section 3.4.4 shows how relationship types can also have attributes.

3.4.1 Relationship Types, Sets and Instances

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

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i are related in some way in the corresponding miniworld situation. For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department the employee works for. Each relationship instance in the relationship set WORKS_FOR associates one employee entity and one department entity. Figure 3.9 illustrates this example, where each relationship instance r_i is shown connected to the employee and department entities that participate in r_i. In the miniworld represented by Figure 3.9, employees e_1, e_3, and e_6 work for department d_1; e_2 and e_4 work for d_2; and e_5 and e_7 work for d_3.

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

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type. The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary,** and one of degree three is called **ternary.**

Petitioner Uber Ex-1052, 0076

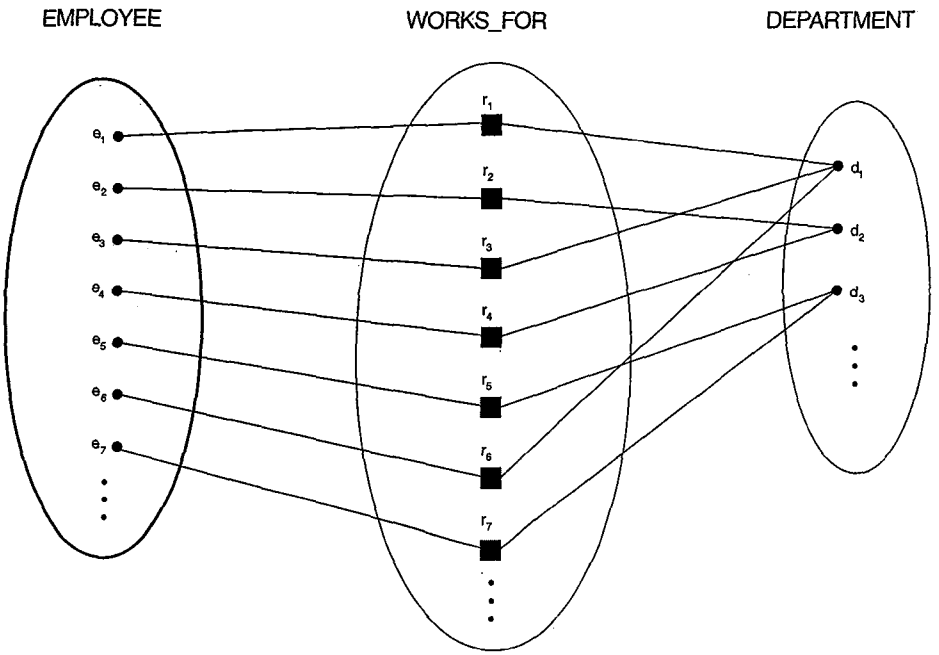


Figure 3.9 Some instances of the WORKS_FOR relationship between EMPLOYEE and DEPARTMENT.

An example of a ternary relationship is SUPPLY, shown in Figure 3.10, where each relationship instance r_i associates three entities—a supplier s, a part p, and a project j—whenever s supplies part p to project j. Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships, and we shall characterize them further in Chapter 4.

Relationships as Attributes. It is sometimes convenient to think of a relationship type in terms of attributes, as we discussed in Section 3.3.3. Consider the WORKS_FOR relationship type of Figure 3.9. One can think of an attribute called Department of the EMPLOYEE entity type whose value for each employee entity is (a reference to) the *department entity* that the employee works for. Hence, the value set for this Department attribute is the *set of all DEPARTMENT entities*. This is what we did in Figure 3.8 when we specified the initial design of the entity type EMPLOYEE for the COMPANY database. However, when we think of a binary relationship as an attribute, we always have two options. In this example, the alternative is to think of a multivalued attribute Employees of the entity type DEPARTMENT whose values for each department entity is the *set of employee entities* who work for that department. The value set of this Employees attribute is the EMPLOYEE entity set. Either of these two

Petitioner Uber Ex-1052, 0077

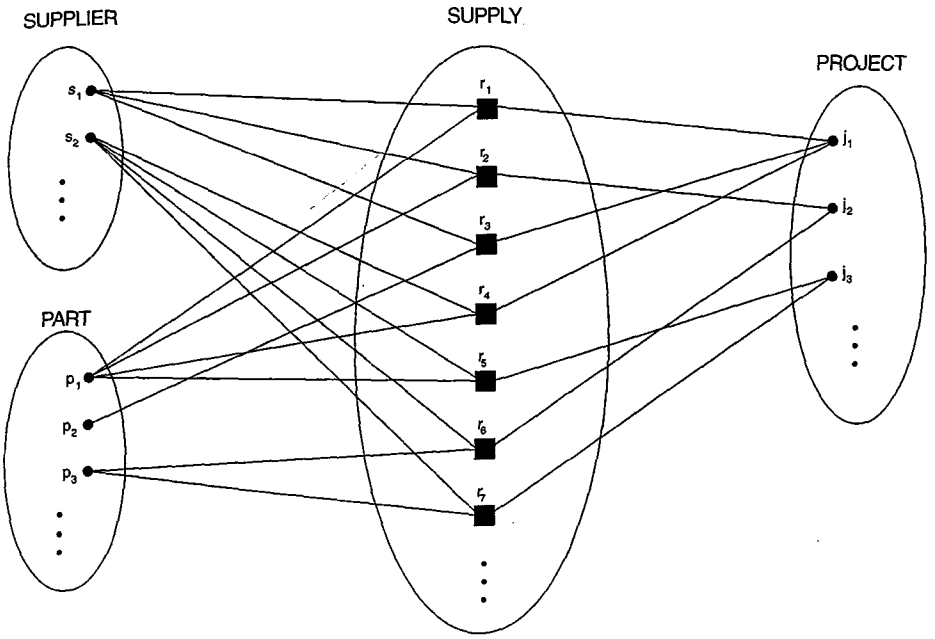


Figure 3.10 Some relationship instances of a ternary relationship SUPPLY.

attributes—Department of EMPLOYEE or Employees of DEPARTMENT—can represent the WORKS_FOR relationship type. If both are represented, they are constrained to be inverses of each other.[7]

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

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for

7. This concept of representing relationship types as attributes is used in a class of data models called **functional data models.** In object databases (see Chapters 11 and 12), relationships can be represented by reference attributes, either in one direction or in both directions as inverses. In relational databases (see Chapters 7 and 8), foreign keys are a type of reference attribute used to represent relationships.

Petitioner Uber Ex-1052, 0078

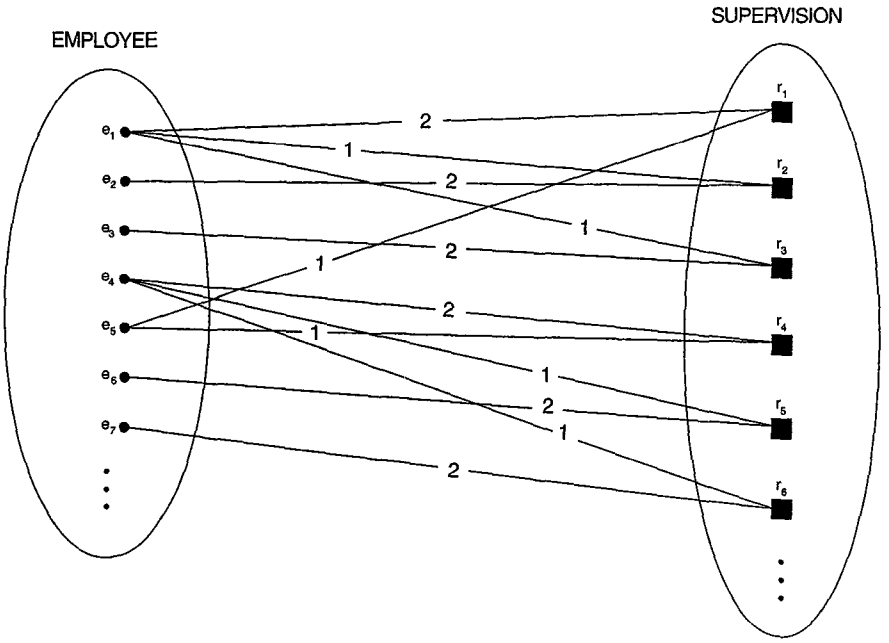


Figure 3.11 The recursive relationship SUPERVISION, where the EMPLOYEE entity type plays the two roles of supervisor (1) and supervisee (2).

distinguishing the meaning of each participation. Such relationship types are called **recursive relationships,** and Figure 3.11 shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or boss), and once in the role of *supervisee* (or subordinate). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of supervisee. In Figure 3.11, the lines marked "1" represent the supervisor role, and those marked "2" represent the supervisee role; hence, e_1 supervises e_2 and e_3; e_4 supervises e_6 and e_7; and e_5 supervises e_1 and e_4.

3.4.3 Constraints on Relationship Types

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

Petitioner Uber Ex-1052, 0079

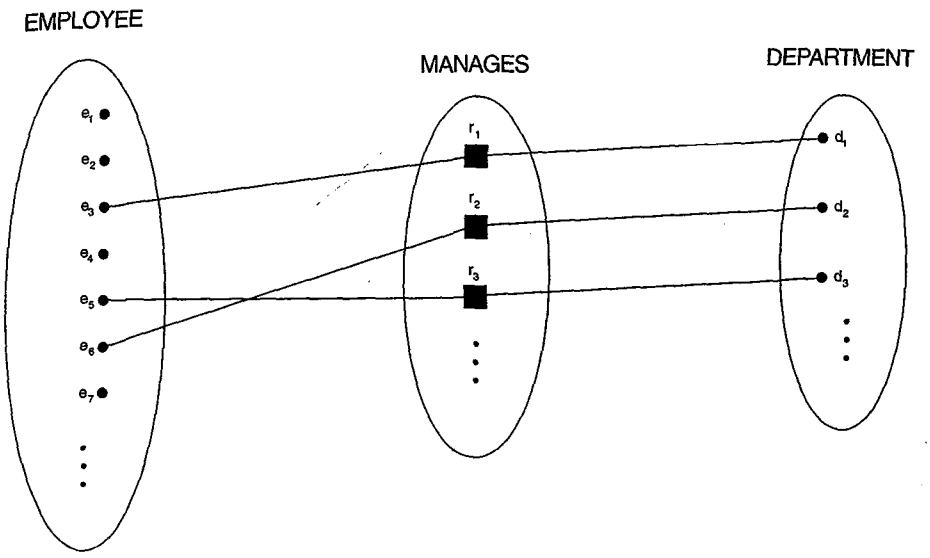


Figure 3.12 The 1:1 relationship MANAGES, with partial participation of EMPLOYEE and total participation of DEPARTMENT.

Cardinality Ratios for Binary Relationships. The **cardinality ratio** for a binary relationship specifies the number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to (that is, employs) numerous employees,[8] but an employee can be related to (work for) only one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES (Figure 3.12), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that an employee can manage only one department and that a department has only one manager. The relationship type WORKS_ON (Figure 3.13) is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.

Cardinality ratios for binary relationships are displayed on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 3.2.

Participation Constraints and Existence Dependencies. The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. There are two types of participation constraints—total and partial—which we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in a

8. N stands for *any number* of related entities (zero or more).

Petitioner Uber Ex-1052, 0080

WORKS_FOR relationship instance (Figure 3.9). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation,** meaning that every entity in "the total set" of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency.** In Figure 3.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial,** meaning that *some* or "part of the set of" employee entities are related to a department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 3.2).

3.4.4 Attributes of Relationship Types

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

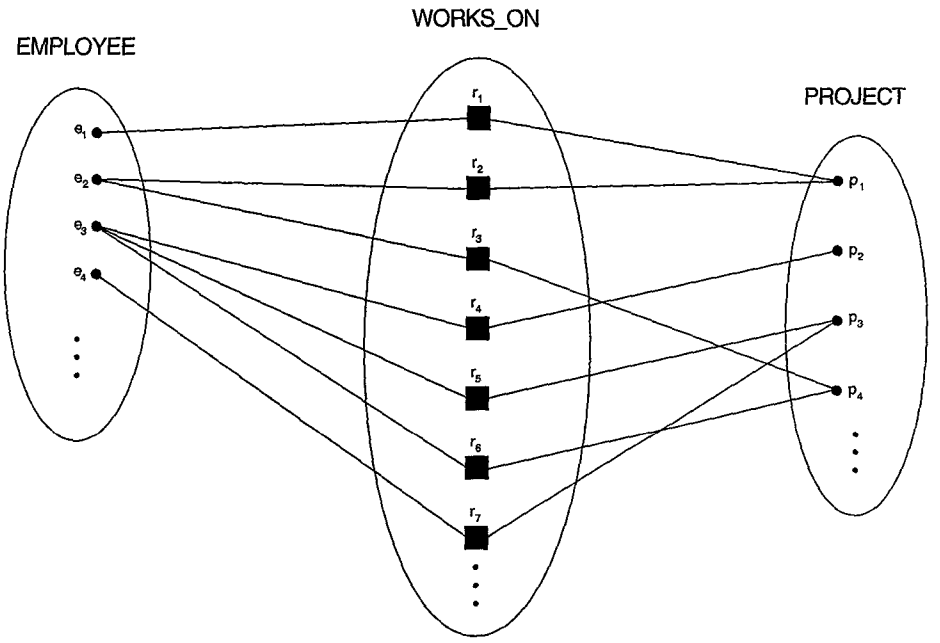


Figure 3.13 The M:N relationship WORKS_ON between EMPLOYEE and PROJECT.

Petitioner Uber Ex-1052, 0081

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

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

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

3.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types.** In contrast, **regular entity types** that do have a key attribute are sometimes called **strong entity types.** Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with some of their attribute values. We call this other entity type the **identifying** or **owner entity type,**[9] and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.[10] A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship, because a weak entity cannot be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (LicenseNumber) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 3.2). The attributes of DEPENDENT are Name (the first name of the dependent), BirthDate, Sex, and Relationship

9. The identifying entity type is also sometimes called the **parent entity type** or the **dominant entity type.**

10. The weak entity type is also sometimes called the **child entity type** or the **subordinate entity type.**

Petitioner Uber Ex-1052, 0082

(to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, BirthDate, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to **own** the dependent entities that are related to it.

A weak entity type normally has a **partial key,** which is the set of attributes that can uniquely identify weak entities that are *related to the same owner entity.*[11] In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

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

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a composite attribute with component attributes Name, BirthDate, Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the weak entity type representation if there are many attributes. If the weak entity participates independently in relationship types other than its identifying relationship type, then it should *not* be modeled as a complex attribute.

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we shall illustrate in Chapter 4.

3.6 Refining the ER Design for the COMPANY Database

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

In our example, we specify the following relationship types:

1. MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.[12] The attribute StartDate is assigned to this relationship type.

11. The partial key is sometimes called the **discriminator.**

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

Petitioner Uber Ex-1052, 0083

2. WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
3. CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users.
4. SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
5. WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
6. DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

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

3.7 ER Diagrams, Naming Conventions, and Design Issues

3.7.1 Summary of Notation for ER Diagrams

Figures 3.9 through 3.13 illustrate the entity types and relationship types by displaying their extensions—the individual entities and relationship instances. In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful because a database schema changes rarely, whereas the extension changes frequently. In addition, the schema is usually easier to display than the extension of a database, because it is much smaller.

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

Petitioner Uber Ex-1052, 0084

illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the NumberOfEmployees attribute of DEPARTMENT.

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

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

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

Figure 3.14 summarizes the conventions for ER diagrams.

3.7.2 Proper Naming of Schema Constructs

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

As a general practice, given a narrative description of the database requirements, the *nouns* appearing in the narrative tend to give rise to entity type names, and the *verbs* tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.

Another naming consideration involves choosing relationship names to make the ER diagram of the schema readable from left to right and from top to bottom. We have generally followed this guideline in Figure 3.2. One exception is the DEPENDENTS_OF relationship type, which reads from bottom to top. This is because we say that the DEPENDENT entities (bottom entity type) are DEPENDENTS_OF (relationship name) an EMPLOYEE (top entity type). To change this to read from top to bottom, we could rename the relationship type to HAS_DEPENDENTS, which would then read: an EMPLOYEE entity (top entity type) HAS_DEPENDENTS (relationship name) of type DEPENDENT (bottom entity type).

Petitioner Uber Ex-1052, 0085

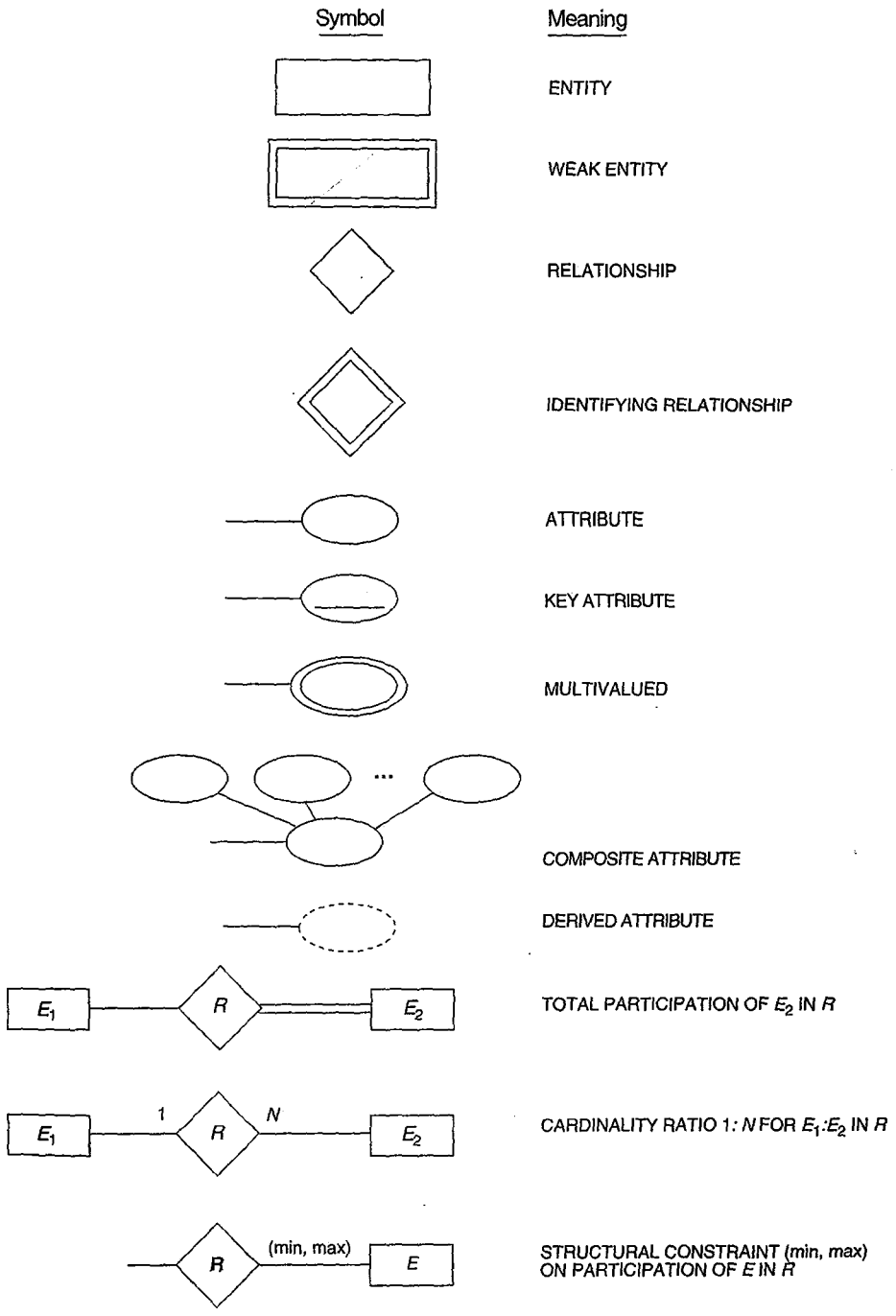


Figure 3.14 Summary of ER diagram notation.

Petitioner Uber Ex-1052, 0086

3.7.3 Design Choices for ER Conceptual Design

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

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

1. A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is often the case that a pair of such attributes that are inverses of one another are refined into a binary relationship. We discussed this type of refinement in detail in Section 3.6.
2. Similarly, an attribute that exists in several entity types may be refined into its own independent entity type. For example, suppose that several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE each have an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute DeptName and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships. Other attributes/relationships of DEPARTMENT may be discovered later.
3. An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute DeptName and related to only one other entity type STUDENT. In this case, DEPARTMENT may be refined into an attribute of STUDENT.
4. In Chapter 4, we will discuss other refinements concerning specialization/generalization and relationships of higher degree.

3.7.4 Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. Appendix A gives some of the more popular notations. In Chapter 4, we will also introduce the Universal Modeling Language (UML) notation, which has been proposed as a standard for conceptual object modeling.

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

Petitioner Uber Ex-1052, 0087

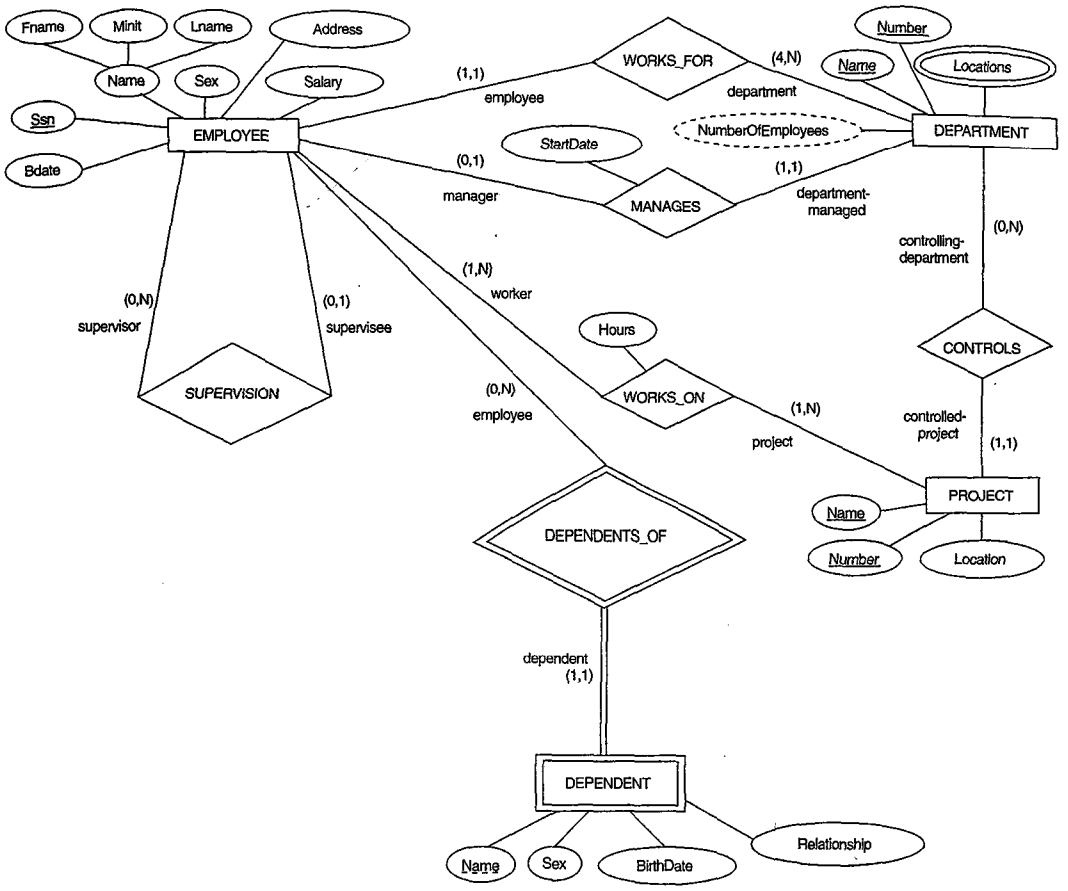


Figure 3.15 ER diagram for the COMPANY schema, with all role names included and with structural constraints on relationships specified using the alternate notation (min, max).

Figure 3.15 displays the COMPANY database schema using the (min, max) notation.[13] Usually, one uses either the cardinality ratio/single line/double line notation *or* the min/max notation. The min/max notation is more precise, and we can use it easily to specify structural constraints for relationship types of *any degree*. However, it is not sufficient for specifying some key constraints on higher degree relationships, as we shall discuss in Chapter 4.

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

13. In some notations, particularly those used in object modeling, the placing of the (min, max) is on the *opposite sides* to the ones we have shown. For example, for the WORKS_FOR relationship in Figure 3.15, the (1,1) would be on the DEPARTMENT side and the (4,N) would be on the EMPLOYEE side. We used the original notation from Abrial (1974).

Petitioner Uber Ex-1052, 0088

3.8 Summary

In this chapter we presented the modeling concepts of a high-level conceptual data model, the Entity-Relationship (ER) model. We started by discussing the role that a high-level data model plays in the database design process, and then we presented an example set of database requirements for the COMPANY database, which is one of the examples that is used throughout this book. We then defined the basic ER model concepts of entities and their attributes. We discussed null values and presented the various types of attributes, which can be nested arbitrarily to produce complex attributes:

- Simple or atomic
- Composite
- Multivalued

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

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

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

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

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

Entity-Relationship schemas can be represented diagrammatically as ER diagrams. We showed how to design an ER schema for the COMPANY database by first defining the entity types and their attributes and then refining the design to include relationship types. We displayed the ER diagram for the COMPANY database schema.

The ER modeling concepts we have presented thus far—entity types, relationship types, attributes, keys, and structural constraints—can model traditional business data-processing database applications. However, many newer, more complex applications—such as engineering design, medical information systems, or telecommunications—require additional concepts if we want to model them with greater accuracy. We will discuss these advanced modeling concepts in Chapter 4. We will also describe ternary and higher-degree relationship types in more detail in Chapter 4, and discuss the circumstances under which they are distinguished from binary relationships.

Petitioner Uber Ex-1052, 0089

Review Questions

3.1. Discuss the role of a high-level data model in the database design process.
3.2. List the various cases where use of a null value would be appropriate.
3.3. Define the following terms: *entity*, *attribute*, *attribute value*, *relationship instance*, *composite attribute*, *multivalued attribute*, *derived attribute*, *complex attribute*, *key attribute*, *value set (domain)*.
3.4. What is an entity type? What is an entity set? Explain the differences among an entity, an entity type, and an entity set.
3.5. Explain the difference between an attribute and a value set.
3.6. What is a relationship type? Explain the differences among a relationship instance, a relationship type, and a relationship set.
3.7. What is a participation role? When is it necessary to use role names in the description of relationship types?
3.8. Describe the two alternatives for specifying structural constraints on relationship types. What are the advantages and disadvantages of each?
3.9. Under what conditions can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity types?
3.10. When we think of relationships as attributes, what are the value sets of these attributes? What class of data models is based on this concept?
3.11. What is meant by a recursive relationship type? Give some examples of recursive relationship types.
3.12. When is the concept of a weak entity used in data modeling? Define the terms *owner entity type*, *weak entity type*, *identifying relationship type*, and *partial key*.
3.13. Can an identifying relationship of a weak entity type be of a degree greater than two? Give examples to illustrate your answer.
3.14. Discuss the conventions for displaying an ER schema as an ER diagram.
3.15. Discuss the naming conventions used for ER schema diagrams.

Exercises

3.16. Consider the following set of requirements for a university database that is used to keep track of students' transcripts. This is similar but not identical to the database shown in Figure 1.2:
   a. The university keeps track of each student's name, student number, social security number, current address and phone, permanent address and phone, birthdate, sex, class (freshman, sophomore, . . ., graduate), major department, minor department (if any), and degree program (B.A., B.S., . . ., Ph.D.). Some user applications need to refer to the city, state, and zip code of the student's permanent address and to the student's last name. Both social security number and student number have unique values for each student.
   b. Each department is described by a name, department code, office number, office phone, and college. Both name and code have unique values for each department.

Petitioner Uber Ex-1052, 0090

c. Each course has a course name, description, course number, number of semester hours, level, and offering department. The value of course number is unique for each course.
d. Each section has an instructor, semester, year, course, and section number. The section number distinguishes sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, . . ., up to the number of sections taught during each semester.
e. A grade report has a student, section, letter grade, and numeric grade (0, 1, 2, 3, or 4).

Design an ER schema for this application, and draw an ER diagram for that schema. Specify key attributes of each entity type and structural constraints on each relationship type. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

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

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

3.19. Consider the ER diagram of Figure 3.16, which shows a simplified schema for an airline reservations system. Extract from the ER diagram the requirements and constraints that produced this schema. Try to be as precise as possible in your requirements and constraints specification.

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

3.21. Design an ER schema for keeping track of information about votes taken in the U.S. House of Representatives during the current two-year congressional session. The database needs to keep track of each U.S. STATE's Name (e.g., Texas, New York, California) and includes the Region of the state (whose domain is {Northeast, Midwest, Southeast, Southwest, West}). Each CONGRESSPERSON in the House of Representatives is described by their Name, and includes the District represented, the StartDate when they were first elected, and the political Party they belong to (whose domain is {Republican, Democrat, Independent, Other}). The database keeps track of each BILL (i.e., proposed law), and includes the BillName, the DateOfVote on the bill, whether the bill PassedOrFailed (whose domain is {YES, NO}), and the Sponsor (the congressperson(s) who sponsored—i.e., proposed—the bill). The database keeps track of how each congressperson voted on each bill

Petitioner Uber Ex-1052, 0091

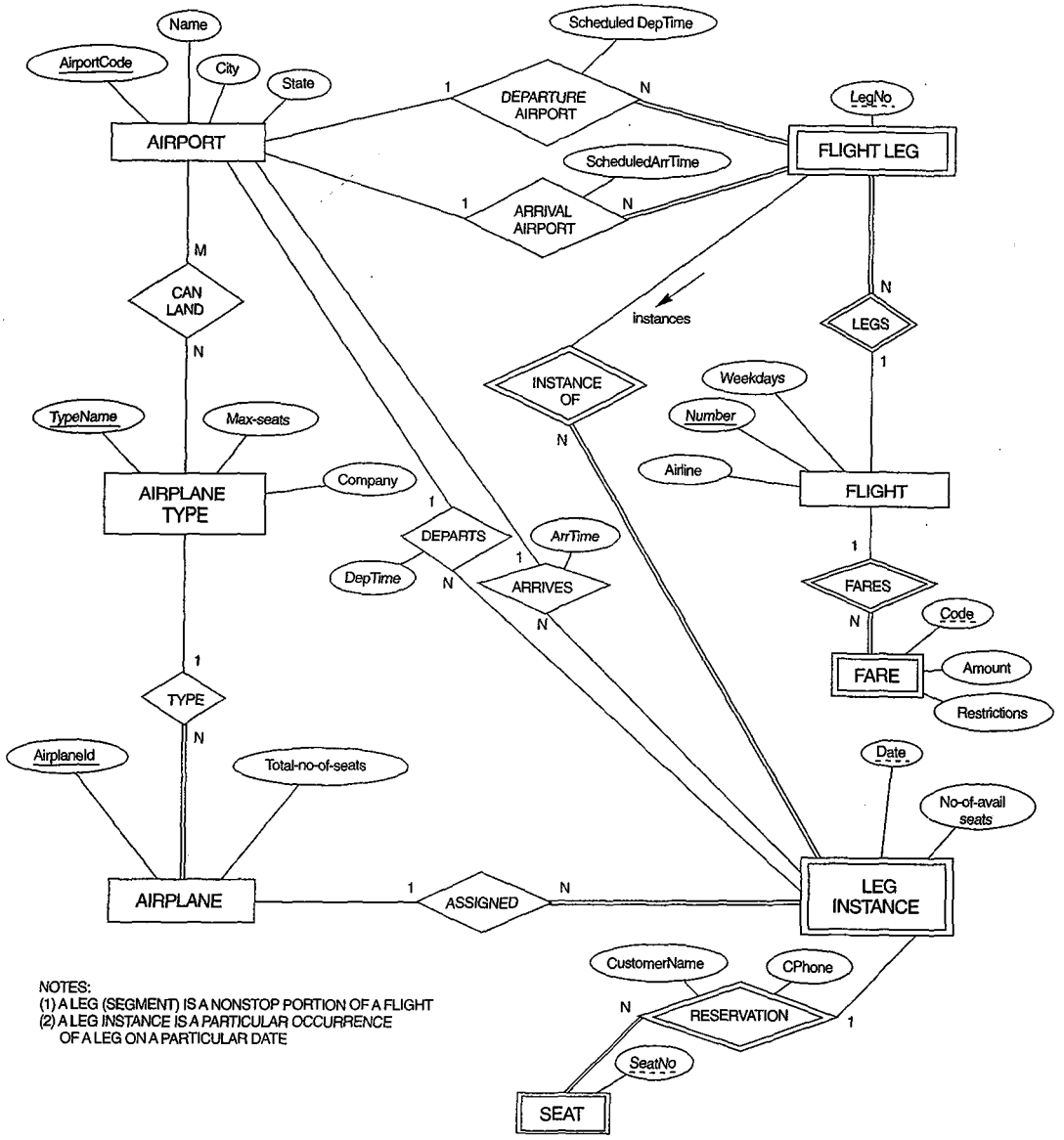


Figure 3.16 An ER diagram for an AIRLINE database.

(domain of vote attribute is {Yes, No, Abstain, Absent}). Draw an ER schema diagram for the above application. State clearly any assumptions you make.

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

Petitioner Uber Ex-1052, 0092

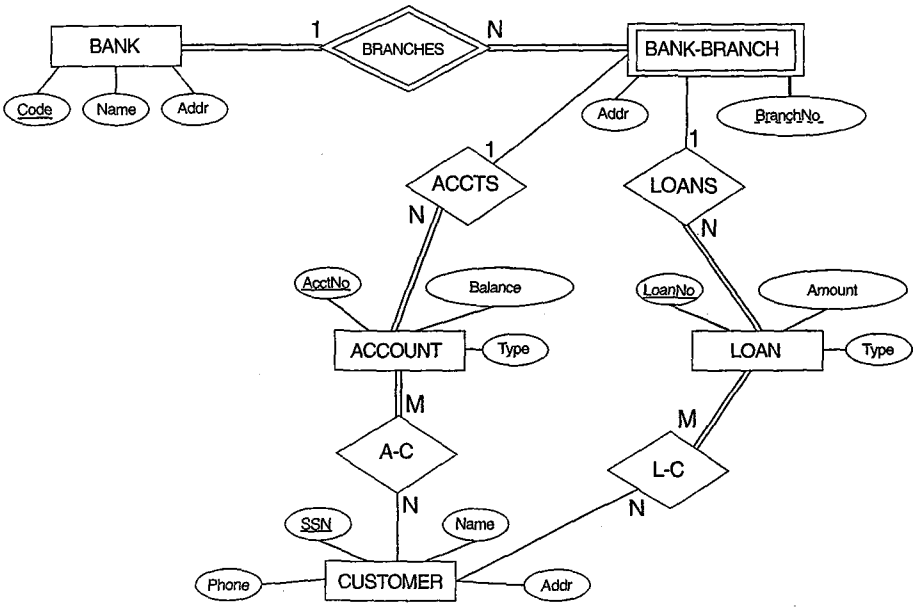


Figure 3.17 An ER diagram for a BANK database.

the positions they played in that game, and the result of the game. Try to design an ER schema diagram for this application, stating any assumptions you make. Choose your favorite sport (soccer, baseball, football, . . .).

3.23. Consider the ER diagram shown in Figure 3.17 for part of a BANK database. Each bank can have multiple branches, and each branch can have multiple accounts and loans.
   a. List the (nonweak) entity types in the ER diagram.
   b. Is there a weak entity type? If so, give its name, partial key, and identifying relationship.
   c. What constraints do the partial key and the identifying relationship of the weak entity type specify in this diagram?
   d. List the names of all relationship types, and specify the (min, max) constraint on each participation of an entity type in a relationship type. Justify your choices.
   e. List concisely the user requirements that led to this ER schema design.
   f. Suppose that every customer must have at least one account but is restricted to at most two loans at a time, and that a bank branch cannot have more than 1000 loans. How does this show up on the (min, max) constraints?

3.24. Consider the ER diagram in Figure 3.18. Assume that an employee may work in up to two departments, but may also not be assigned to any department. Assume that each department must have one and may have up to three phone numbers. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions*

Petitioner Uber Ex-1052, 0093

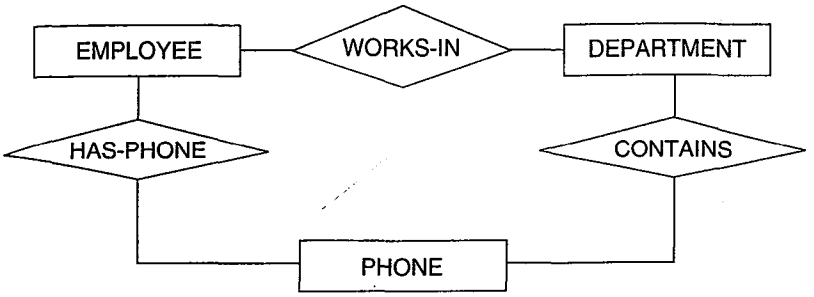


Figure 3.18 An ER diagram for a database that keeps track of company and employee phones.

you make. Under what conditions would the relationship HAS_PHONE be redundant in the above example?

3.25. Consider the ER diagram in Figure 3.19. Assume that a course may or may not use a textbook, but that a text by definition is a book that is used in some course. A course may not use more than five books. Instructors teach from two to four courses. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make*. If we add the relationship ADOPTS between INSTRUCTOR and TEXT, what (min, max) constraints would you put on it? Why?

3.26. Consider an entity type SECTION in a UNIVERSITY database, which describes the section offerings of courses. The attributes of SECTION are: SectionNumber, Semester, Year, CourseNumber, Instructor, RoomNo (where section is taught), Building (where section is taught), Weekdays (domain is the possible combinations of weekdays in which a section can be offered {MWF, MW, TT, etc.}), and Hours

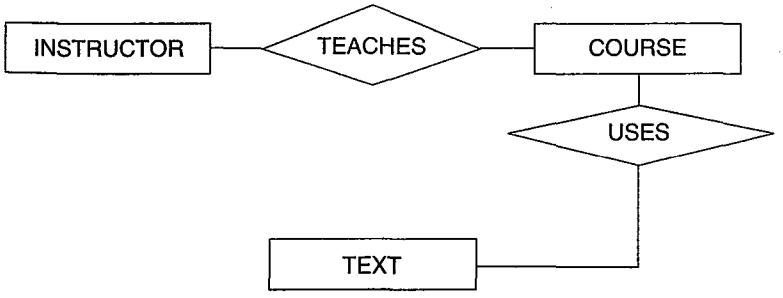


Figure 3.19 An ER diagram for a database that keeps track of textbooks used in courses.

Petitioner Uber Ex-1052, 0094

(domain is all possible time periods during which sections are offered {9–9.50 A.M., 10–10.50 A.M., . . ., 3.30–4.50 P.M., 5.30–6.20 P.M., etc.}). Assume that SectionNumber is unique for each course within a particular semester/year combination (that is, if a course is offered multiple times during a particular semester, its section offerings are numbered 1, 2, 3, etc.). There are several composite keys for SECTION, and some attributes are components of more than one key. Identify three composite keys, and show how they can be represented in an ER schema diagram.

Selected Bibliography

The Entity-Relationship model was introduced by Chen (1976), and related work appears in Schmidt and Swenson (1975), Wiederhold and Elmasri (1979), and Senko (1975). Since then, numerous modifications to the ER model have been suggested. We have incorporated some of these in our presentation. Structural constraints on relationships are discussed in Abrial (1974), Elmasri and Wiederhold (1980), and Lenzerini and Santucci (1983). Multivalued and composite attributes are incorporated in the ER model in Elmasri et al. (1985). Although we did not discuss languages for the entity-relationship model and its extensions, there have been several proposals for such languages. Elmasri and Wiederhold (1981) propose the GORDAS query language for the ER model. Another ER query language is proposed by Markowitz and Raz (1983). Senko (1980) presents a query language for Senko's DIAM model. A formal set of operations called the ER algebra was presented by Parent and Spaccapietra (1985). Gogolla and Hohenstein (1991) present another formal language for the ER model. Campbell et al. (1985) present a set of ER operations and show that they are relationally complete. A conference for the dissemination of research results related to the ER model has been held regularly since 1979. The conference, now known as the International Conference on Conceptual Modeling, has been held in Los Angeles (ER 1979, ER 1983, ER 1997), Washington (ER 1981), Chicago (ER 1985), Dijon, France (ER 1986), New York City (ER 1987), Rome (ER 1988), Toronto (ER 1989), Lausanne, Switzerland (ER 1990), San Mateo, California (ER 1991), Karlsruhe, Germany (ER 1992), Arlington, Texas (ER 1993), Manchester, England (ER 1994), Brisbane, Australia (ER 1995), Cottbus, Germany (ER 1996), and Singapore (ER 1998).

Petitioner Uber Ex-1052, 0095

CHAPTER 4 Enhanced Entity-Relationship and Object Modeling

The ER modeling concepts discussed in Chapter 3 are sufficient for representing many database schemas for "traditional" database applications, which mainly include data-processing applications in business and industry. Since the late 1970s, however, newer applications of database technology have become commonplace; these include databases for engineering design and manufacturing (CAD/CAM[1]), telecommunications, images and graphics, multimedia[2], data mining, data warehousing, geographic information systems (GIS), and databases for indexing the World Wide Web, among many other applications. These types of databases have more complex requirements than do the more traditional applications. To represent these requirements as accurately and clearly as possible, designers of database applications must use additional *semantic data modeling* concepts. Various semantic data models have been proposed in the literature.

In this chapter, we describe features that have been proposed for semantic data models, and show how the ER model can be enhanced to include these concepts, leading to the **enhanced-ER** or **EER** model.[3] We start in Section 4.1 by incorporating the concepts of *class/subclass relationships* and *type inheritance* into the ER model. Then, in Section 4.2, we add the concepts of *specialization* and *generalization*. Section 4.3 discusses *constraints* on

1. This stands for computer-aided design/computer-aided manufacturing.
2. These store multimedia data, such as pictures, voice messages, and video clips.
3. EER has also been used to stand for *extended* ER model.

Petitioner Uber Ex-1052, 0096

specialization/generalization, and Section 4.4 shows how the UNION construct can be modeled by including the concept of *category* in the EER model. Section 4.5 gives an example UNIVERSITY database schema in the EER model, and summarizes the EER model concepts by giving formal definitions.

The object data model (see Chapters 11 and 12) includes many of the concepts proposed for semantic data models. Object modeling methodologies, such as OMT (Object Modeling Technique) and UML (Universal Modeling Language) are becoming increasingly popular in software design and engineering. These methodologies go beyond database design to specify detailed design of software modules and their interactions using various types of diagrams. An important part of these methodologies—namely, the *class diagrams*[4]—are similar in many ways to EER diagrams. However, in addition to specifying attributes and relationships in class diagrams, the *operations* on objects are also specified. Operations can be used to specify the *functional requirements* during database design, as we discussed in Section 3.1 and illustrated in Figure 3.1. We will present the UML notation and concepts for class diagrams in Section 4.6, and briefly compare these to EER notation and concepts.

Section 4.7 discusses some of the more complex issues involved in modeling of ternary and higher-degree relationships. In Section 4.8, we discuss the fundamental abstractions that are used as the basis of many semantic data models. Section 4.9 summarizes the chapter.

For a detailed introduction to conceptual modeling, Chapter 4 should be considered a continuation of Chapter 3. However, if only a basic introduction to ER modeling is desired, this chapter may be omitted. Alternatively, the reader may choose to skip some or all of the later sections of this chapter (Sections 4.3 through 4.8).

4.1 Subclasses, Superclasses, and Inheritance

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

The first EER model concept we take up is that of a **subclass** of an entity type. As we discussed in Chapter 3, an entity type is used to represent both a *type of entity,* and the *entity set* or *collection of entities of that type* that exist in the database. For example, the

4. A **class** is similar to an *entity type* in many ways.

Petitioner Uber Ex-1052, 0097

entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database. In many cases an entity type has numerous subgroupings of its entities that are meaningful and need to be represented explicitly because of their significance to the database application. For example, the entities that are members of the EMPLOYEE entity type may be grouped further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on. The set of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these subgroupings is also an employee. We call each of these subgroupings a **subclass** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** for each of these subclasses.

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or simply **class/subclass relationship.**[5] In our previous example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships. Notice that a member entity of the subclass represents the *same real-world entity* as some member of the superclass; for example, a SECRETARY entity 'Joan Logano' is also the EMPLOYEE 'Joan Logano'. Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role*. When we implement a superclass/subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity. In Section 9.2, we discuss various options for representing superclass/subclass relationships in relational databases.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE of the EMPLOYEE entity type. However, it is not necessary that every entity in a superclass be a member of some subclass.

An important concept associated with subclasses is that of **type inheritance.** Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes *as well as* values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass **inherits** all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates. Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass, can be considered an *entity type* in its own right.[6]

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

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

Petitioner Uber Ex-1052, 0098

4.2 Specialization and Generalization

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

Figure 4.1 shows how we represent a specialization diagrammatically in an **EER diagram.** The subclasses that define a specialization are attached by lines to a circle, which is connected to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship.[7] Attributes that apply only to entities of a particular subclass—such as TypingSpeed of SECRETARY—are attached to the rectangle representing that subclass. These are called **specific attributes** (or **local attributes**) of the subclass. Similarly, a subclass can participate in **specific relationship types,** such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO relationship in Figure 4.1. We will explain the **d** symbol in the circles of Figure 4.1 and additional EER diagram notation shortly.

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

There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, the SECRETARY subclass may have an attribute TypingSpeed, whereas the ENGINEER subclass may have an attribute

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

Petitioner Uber Ex-1052, 0099

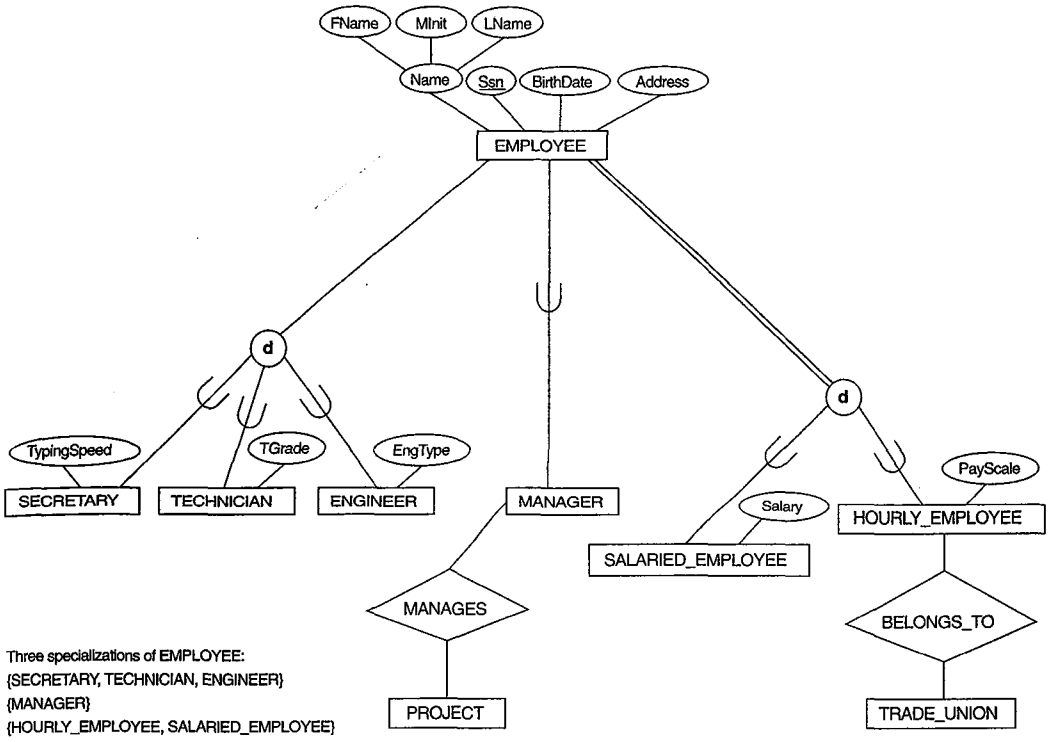


Figure 4.1 EER diagram notation for representing specialization and subclasses.

EngineerType, but SECRETARY and ENGINEER share their other attributes as members of the EMPLOYEE entity type.

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

In summary, the specialization process allows us to do the following:

- Define a set of subclasses of an entity type.
- Establish additional specific attributes with each subclass.
- Establish additional specific relationship types between each subclass and other entity types or other subclasses.

Generalization. We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize**

Petitioner Uber Ex-1052, 0100

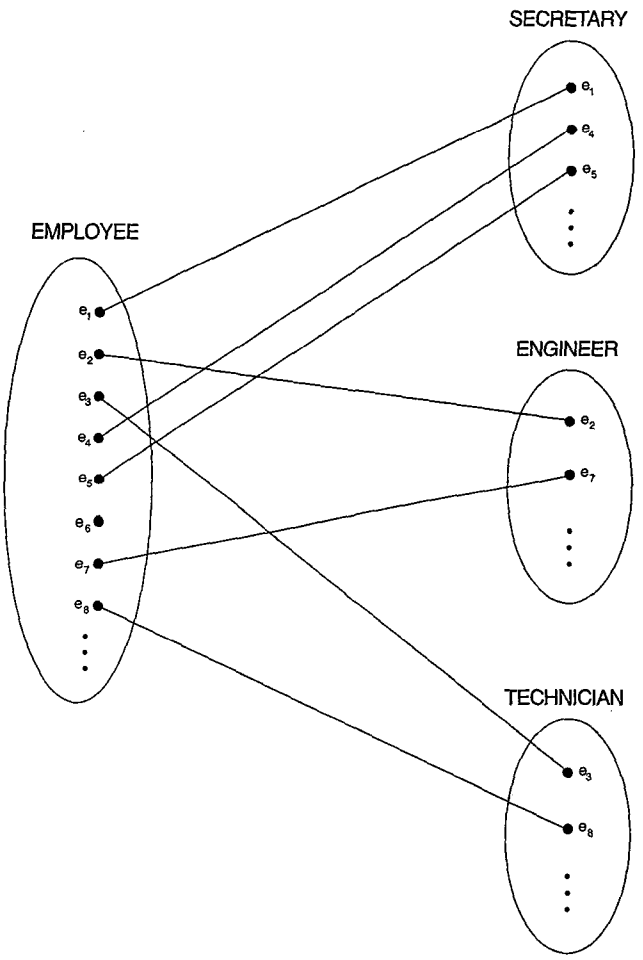


Figure 4.2 Some instances of the specialization of EMPLOYEE into the {SECRETARY, ENGINEER, TECHNICIAN} set of subclasses.

them into a single **superclass** of which the original entity types are special **subclasses.** For example, consider the entity types CAR and TRUCK shown in Figure 4.3(a); they can be generalized into the entity type VEHICLE, as shown in Figure 4.3(b). Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 4.3 we can view {CAR, TRUCK} as a

Petitioner Uber Ex-1052, 0101

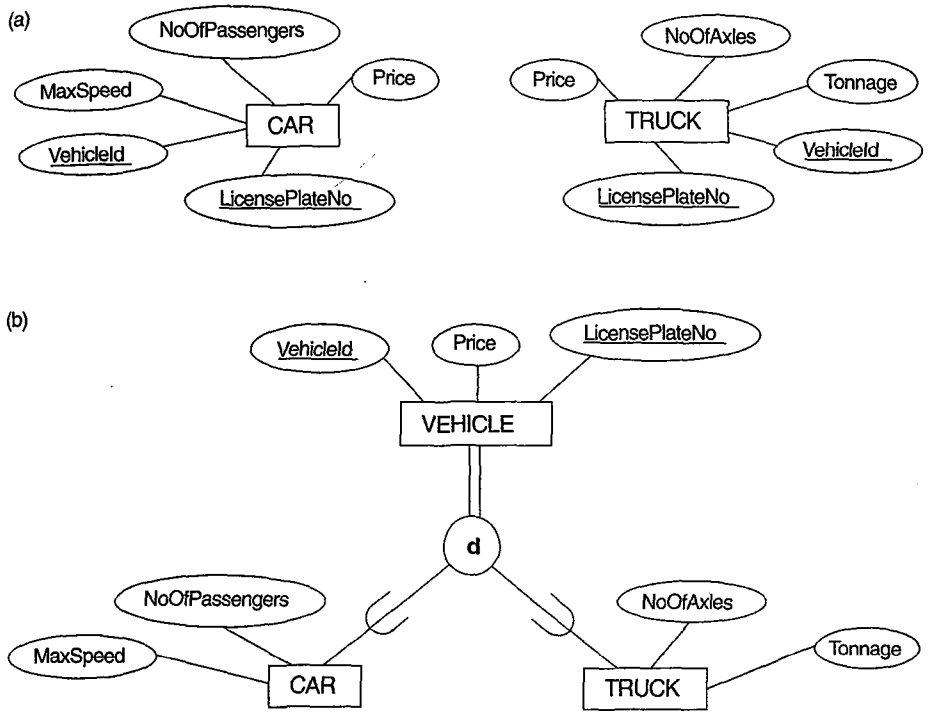


Figure 4.3 Examples of generalization. (a) Two entity types CAR and TRUCK. (b) Generalizing CAR and TRUCK into VEHICLE.

specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 4.1 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will *not* use this notation, because the decision as to which process is more appropriate in a particular situation is often subjective. Appendix A gives some of the suggested alternative diagrammatic notations for schema diagrams/class diagrams.

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

Petitioner Uber Ex-1052, 0102

4.3 Constraints and Characteristics of Specialization and Generalization

In this section, we first discuss constraints that apply to a single specialization or a single generalization; however, for brevity, our discussion refers only to *specialization* even though it applies to *both* specialization and generalization. We then discuss the differences between specialization/generalization *lattices* (*multiple inheritance*) and *hierarchies* (*single inheritance*), and elaborate on the differences between the specialization and generalization processes during conceptual database schema design.

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

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

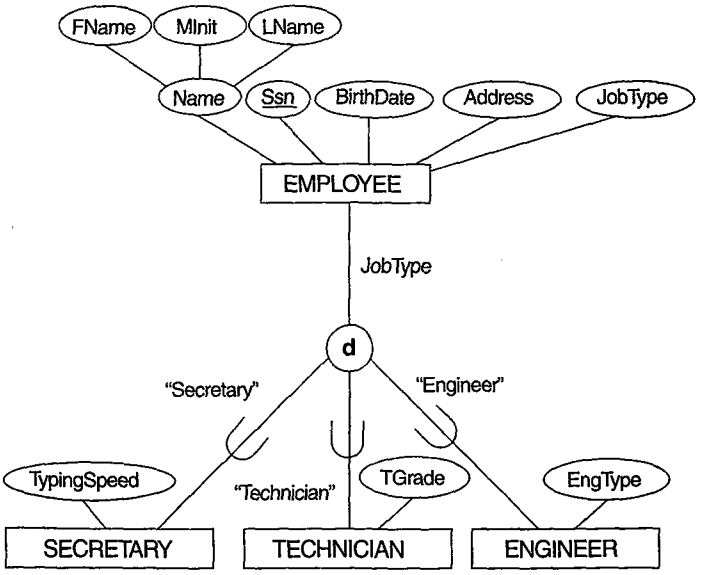


Figure 4.4 An attribute-defined specialization on the JobType attribute of EMPLOYEE.

Petitioner Uber Ex-1052, 0103

all entities of the EMPLOYEE entity type whose attribute value for JobType is 'Secretary' must belong to the subclass. We display a predicate-defined subclass by writing the predicate condition next to the line that connects the subclass to the specialization circle.

If *all* subclasses in a specialization have the membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization,** and the attribute is called the **defining attribute** of the specialization.[8] We display an attribute-defined specialization, as shown in Figure 4.4, by placing the defining attribute name next to the arc from the circle to the superclass.

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

Two other constraints may apply to a specialization. The first is the **disjointness constraint,** which specifies that the subclasses of the specialization must be disjoint. This means that an entity can be a member of *at most one* of the subclasses of the specialization. A specialization that is attribute-defined implies the disjointness constraint if the attribute used to define the membership predicate is single-valued. Figure 4.4 illustrates this case, where the d in the circle stands for disjoint. We also use the d notation to specify the constraint that user-defined subclasses of a specialization must be disjoint, as illustrated by the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 4.1. If the subclasses are not constrained to be disjoint, their sets of entities may **overlap;** that is, the same (real-world) entity may be a member of more than one subclass of the specialization. This case, which is the default, is displayed by placing an o in the circle, as shown in Figure 4.5.

The second constraint on specialization is called the **completeness constraint,** which may be total or partial. A **total specialization** constraint specifies that *every* entity in the superclass must be a member of some subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} of Figure 4.1 is a total specialization of EMPLOYEE; this is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a **partial specialization,** which allows an entity not to belong to any of the subclasses. For example, if some EMPLOYEE entities do not belong to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} of Figures 4.1 and 4.4, then that specialization is partial.[9] Notice that the disjointness and completeness constraints are *independent.* Hence, we have the following four possible constraints on specialization:

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

8. Such an attribute is called a *discriminator* in UML terminology.

9. The notation of using single/double lines is similar to that for partial/total participation of an entity type in a relationship type, as we described in Chapter 3.

Petitioner Uber Ex-1052, 0104

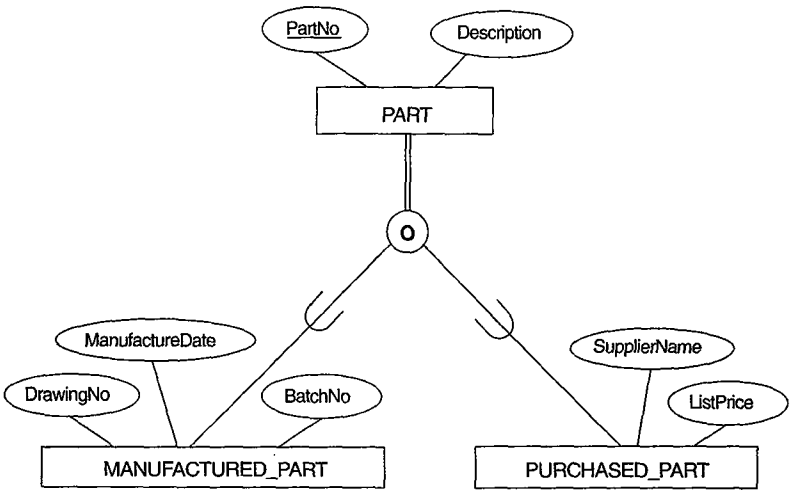


Figure 4.5 Notation for specialization with overlapping (nondisjoint) subclasses.

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

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

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

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

Specialization/Generalization Hierarchies and Lattices. A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specializations. For example, in Figure 4.6 ENGINEER is a subclass of EMPLOYEE and is also a superclass of ENGINEERING_MANAGER; this represents the real-world constraint that every engineering manager is required to be an engineer. A **specialization hierarchy** has the constraint that

Petitioner Uber Ex-1052, 0105

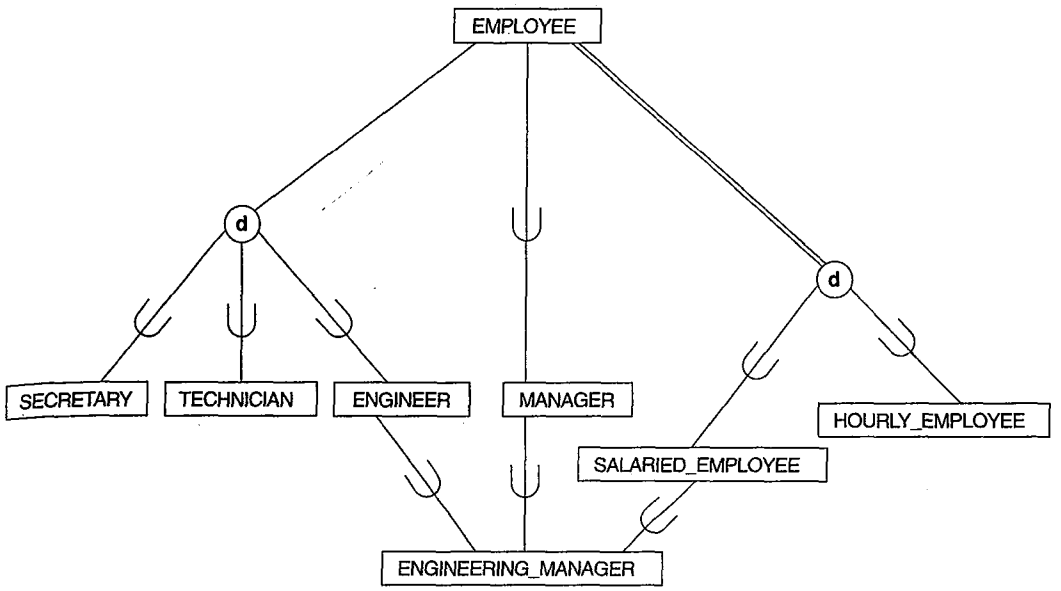


Figure 4.6 A specialization lattice with the shared subclass ENGINEERING_MANAGER.

every subclass participates *as a subclass* in *only one* class/subclass relationship. In contrast, for a **specialization lattice** a subclass can be a subclass in *more than one* class/subclass relationship. Hence, Figure 4.6 is a lattice.

Figure 4.7 shows another specialization lattice of more than one level. This may be part of a conceptual schema for a UNIVERSITY database. Notice that this arrangement would have been a hierarchy except for the STUDENT_ASSISTANT subclass, which is a subclass in two distinct class/subclass relationships. In Figure 4.7, all person entities represented in the database are members of the PERSON entity type, which is specialized into the subclasses {EMPLOYEE, ALUMNUS, STUDENT}. This specialization is overlapping; for example, an alumnus may also be an employee and may also be a student pursuing an advanced degree. The subclass STUDENT is superclass for the specialization {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}, while EMPLOYEE is superclass for the specialization {STUDENT_ASSISTANT, FACULTY, STAFF}. Notice that STUDENT_ASSISTANT is also a subclass of STUDENT. Finally, STUDENT_ASSISTANT is superclass for the specialization into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}.

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

Petitioner Uber Ex-1052, 0106

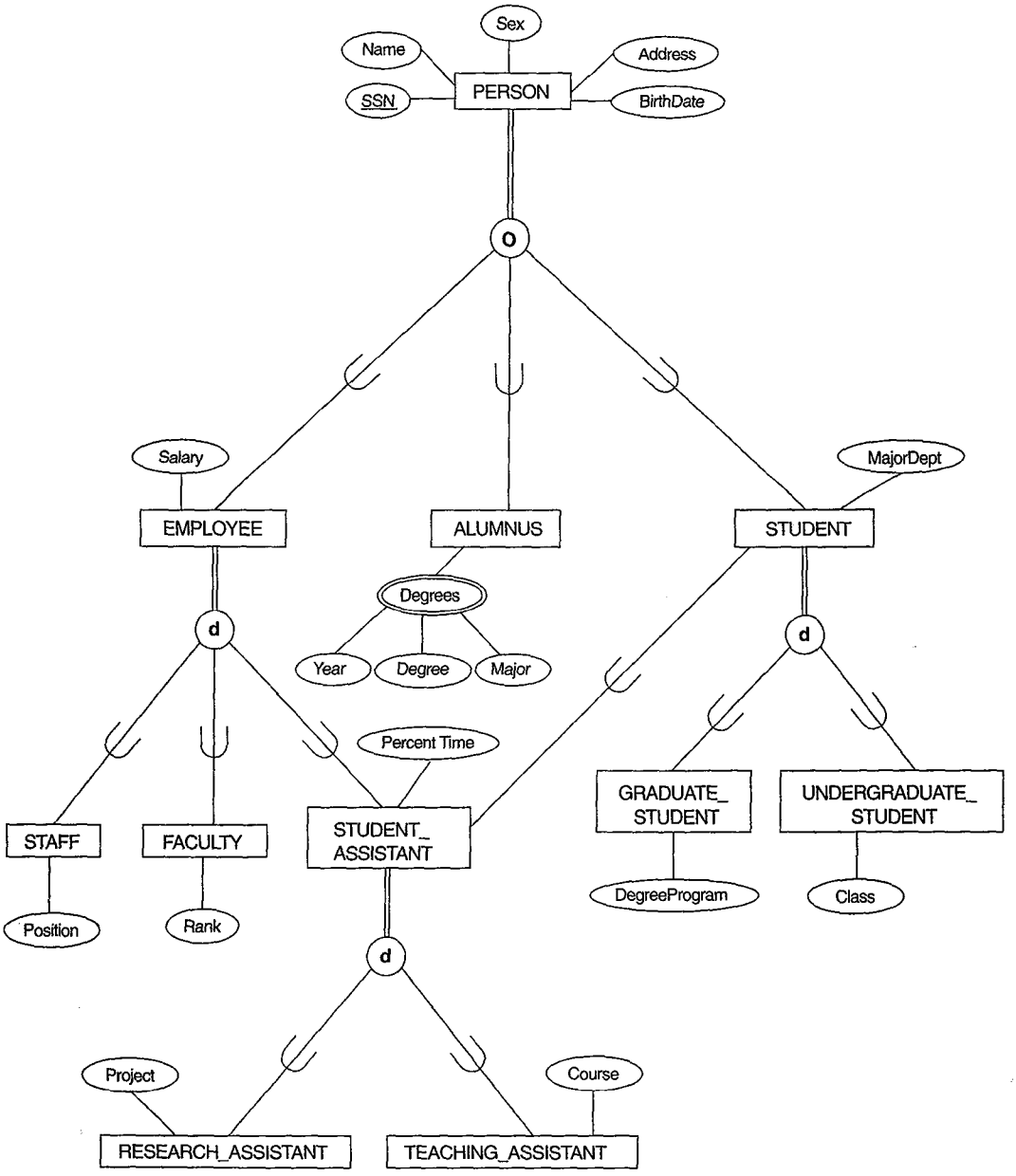


Figure 4.7 A specialization lattice (with multiple inheritance) for a UNIVERSITY database.

Petitioner Uber Ex-1052, 0107

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

It is important to note here that some inheritance mechanisms *do not allow* multiple inheritance (shared subclasses). In such a model, it is necessary to create additional subclasses to cover all possible combinations of classes that may have some entity belong to all these classes simultaneously. Hence, any *overlapping* specialization would require multiple additional subclasses. For example, in the overlapping specialization of PERSON into {EMPLOYEE, ALUMNUS, STUDENT} (or {E, A, S} for short), it would be necessary to create seven subclasses of PERSON: E, A, S, E_A, E_S, A_S, and E_A_S in order to cover all possible types of entities. Obviously, this can lead to extra complexity.

It is also important to note that some inheritance mechanisms that allow multiple inheritance do not allow an entity to have multiple types, and hence an entity can be a member *of only one class.*[10] In such a model, it is also necessary to create additional shared subclasses as leaf nodes to cover all possible combinations of classes that may have some entity belong to all these classes simultaneously. Hence, we would require the same seven subclasses of PERSON.

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

Utilizing Specialization and Generalization in Conceptual Data Modeling. We now elaborate on the differences between the specialization and generalization processes during conceptual database design. In the specialization process, we typically start with an entity type and then define subclasses of the entity type by successive specialization; that is, we repeatedly define more specific groupings of the entity type. For example, when designing the specialization lattice in Figure 4.7, we may first specify an entity type PERSON for a university database. Then we discover that three types of persons will be represented in the database: university employees, alumni, and students. We create the specialization {EMPLOYEE, ALUMNUS, STUDENT} for this purpose and choose the overlapping constraint because a person

10. In some cases, the class is further restricted to be a *leaf node* in the hierarchy or lattice.

Petitioner Uber Ex-1052, 0108

may belong to more than one of the subclasses. We then specialize EMPLOYEE further into {STAFF, FACULTY, STUDENT_ASSISTANT}, and specialize STUDENT into {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. Finally, we specialize STUDENT_ASSISTANT into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}. This successive specialization corresponds to a **top-down conceptual refinement process** during conceptual schema design. So far, we have a hierarchy; we then realize that STUDENT_ASSISTANT is a shared subclass, since it is also a subclass of STUDENT, leading to the lattice.

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

In structural terms, hierarchies or lattices resulting from either process may be identical; the only difference relates to the manner or order in which the schema superclasses and subclasses were specified. In practice, it is likely that neither the generalization process nor the specialization process is followed strictly, but a combination of the two processes is employed. In this case, new classes are continually incorporated into a hierarchy or lattice as they become apparent to users and designers. Notice that the notion of representing data and knowledge by using superclass/subclass hierarchies and lattices is quite common in knowledge-based systems and expert systems, which combine database technology with artificial intelligence techniques. For example, frame-based knowledge representation schemes closely resemble class hierarchies. Specialization is also common in software engineering design methodologies that are based on the object-oriented paradigm.

4.4 Modeling of UNION Types Using Categories

All of the superclass/subclass relationships we have seen thus far have a *single superclass*. A shared subclass such as ENGINEERING_MANAGER in the lattice of Figure 4.6 is the subclass in three *distinct* superclass/subclass relationships, where each of the three relationships has a *single* superclass. It is not uncommon, however, that the need arises for modeling a single superclass/subclass relationship with *more than one* superclass, where the superclasses represent different entity types. In this case, the subclass will represent a collection of objects that is (a subset of) the UNION of distinct entity types; we call such a *subclass* a **union type** or a **category.**[11]

11. Our use of the term *category* is based on the ECR (Entity-Category-Relationship) model (Elmasri et al. 1985).

Petitioner Uber Ex-1052, 0109

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

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

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

It is interesting to note the difference between the category REGISTERED_VEHICLE (Figure 4.8) and the generalized superclass VEHICLE (Figure 4.3(b)). In Figure 4.3(b) every car and every truck is a VEHICLE; but in Figure 4.8 the REGISTERED_VEHICLE category includes some cars and some trucks but not necessarily all of them (for example, some cars or trucks may not be registered). In general, a specialization or generalization such as that in Figure 4.3(b), if it were *partial*, would not preclude VEHICLE from containing other types of entities, such as motorcycles. However, a category such as REGISTERED_VEHICLE in Figure 4.8 implies that only cars and trucks, but not other types of entities, can be members of REGISTERED_VEHICLE.

A category can be **total** or **partial.** For example, ACCOUNT_HOLDER is a predicate-defined partial category in Figure 4.9(a), where c_1 and c_2 are predicate conditions that specify which COMPANY and PERSON entities, respectively, are members of ACCOUNT_HOLDER. However, the category PROPERTY in Figure 4.9(b) is total because every building and lot must be a member of PROPERTY; this is shown by a double line connecting the category and the circle. Partial categories are indicated by a single line connecting the category and the circle, as in Figures 4.8 and 4.9(a).

Petitioner Uber Ex-1052, 0110

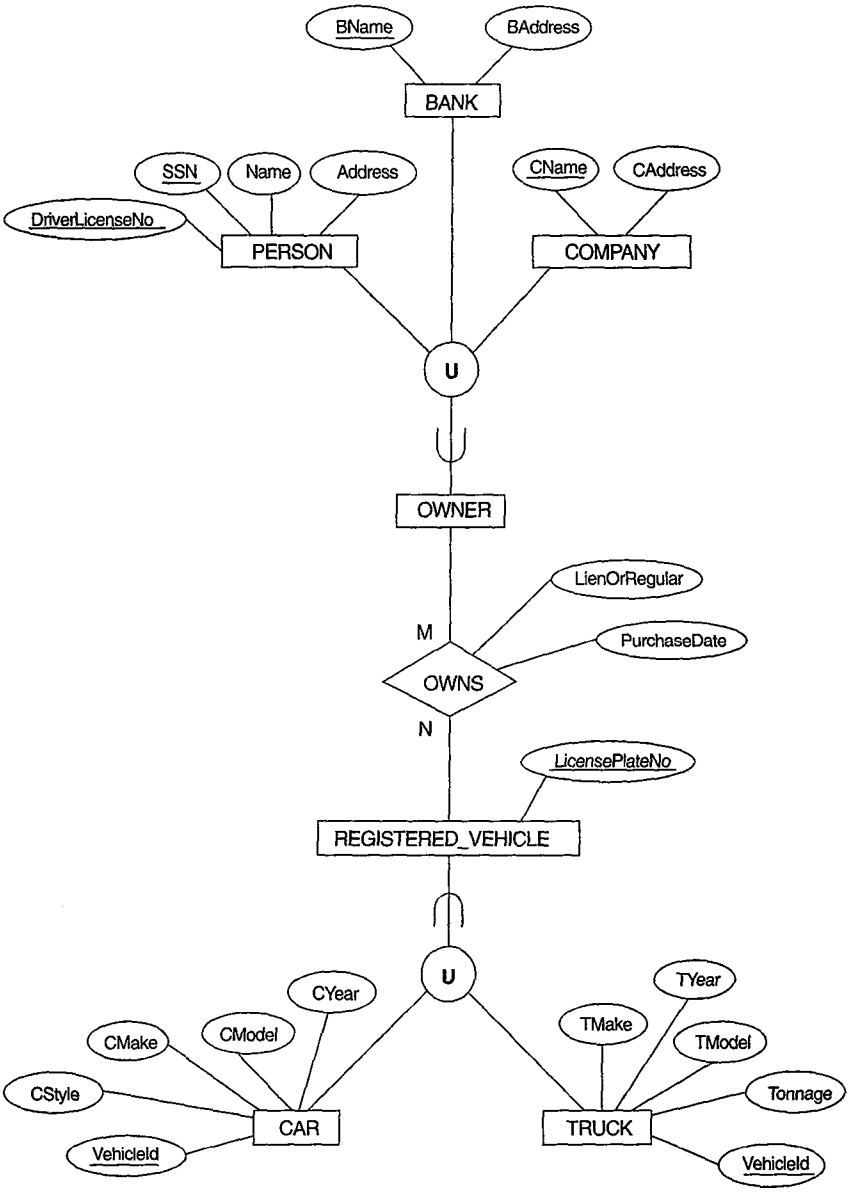


Figure 4.8 An illustration of how to represent the UNION of two or more entity types/classes using the category notation. Two categories are shown: OWNER and REGISTERED_VEHICLE.

Petitioner Uber Ex-1052, 0111

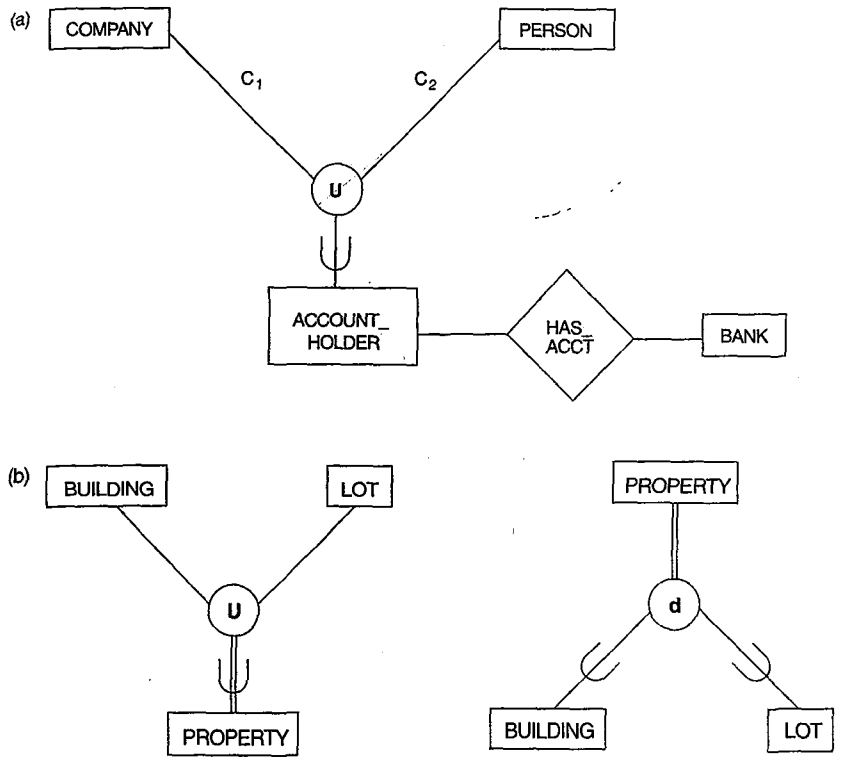


Figure 4.9 Total and partial categories. (a) Partial category ACCOUNT_HOLDER that is a subset of the union of two entity types COMPANY and PERSON. (b) Total category PROPERTY and a similar generalization.

The superclasses of a category may have different key attributes, as demonstrated by the OWNER category of Figure 4.8; or they may have the same key attribute, as demonstrated by the REGISTERED_VEHICLE category. Notice that if a category is total (not partial), it may be represented alternatively as a specialization (or a generalization), as illustrated in Figure 4.9(b). In this case the choice of which representation to use is subjective. If the two classes represent the same type of entities and share numerous attributes, including the same key attributes, specialization/generalization is preferred; otherwise, categorization (union type) is more appropriate.

Petitioner Uber Ex-1052, 0112

4.5 An Example UNIVERSITY EER Schema and Formal Definitions for the EER Model

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

The UNIVERSITY Database Example. For our example database application, consider a UNIVERSITY database that keeps track of students and their majors, transcripts, and registration as well as of the university's course offerings. The database also keeps track of the sponsored research projects of faculty and graduate students. This schema is shown in Figure 4.10. A discussion of the requirements that led to this schema follows.

For each person, the database maintains information on the person's Name [Name], social security number [Ssn], address [Address], sex [Sex], and birth date [BDate]. Two subclasses of the PERSON entity type were identified: FACULTY and STUDENT. Specific attributes of FACULTY are rank [Rank] (assistant, associate, adjunct, research, visiting, etc.), office [FOffice], office phone [FPhone], and salary [Salary], and all faculty members are related to the academic department(s) with which they are affiliated [BELONGS] (a faculty member can be associated with several departments, so the relationship is M:N). A specific attribute of STUDENT is [Class] (freshman = 1, sophomore = 2, . . . , graduate student = 5). Each student is also related to his or her major and minor departments, if known ([MAJOR] and [MINOR]), to the course sections he or she is currently attending [REGISTERED], and to the courses completed [TRANSCRIPT]. Each transcript instance includes the grade the student received [Grade] in the course section.

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

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

A course has attributes course number [C#], course name [Cname], and course description [CDesc]. Several sections of each course are offered, with each section having the attributes section number [Sec#] and the year and quarter in which the section was offered ([Year] and [Qtr]).[12] Section numbers uniquely identify each section. The sections being offered during the current semester are in a subclass CURRENT_SECTION of SECTION, with the defining predicate Qtr = CurrentQtr and Year = CurrentYear. Each

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

Petitioner Uber Ex-1052, 0113

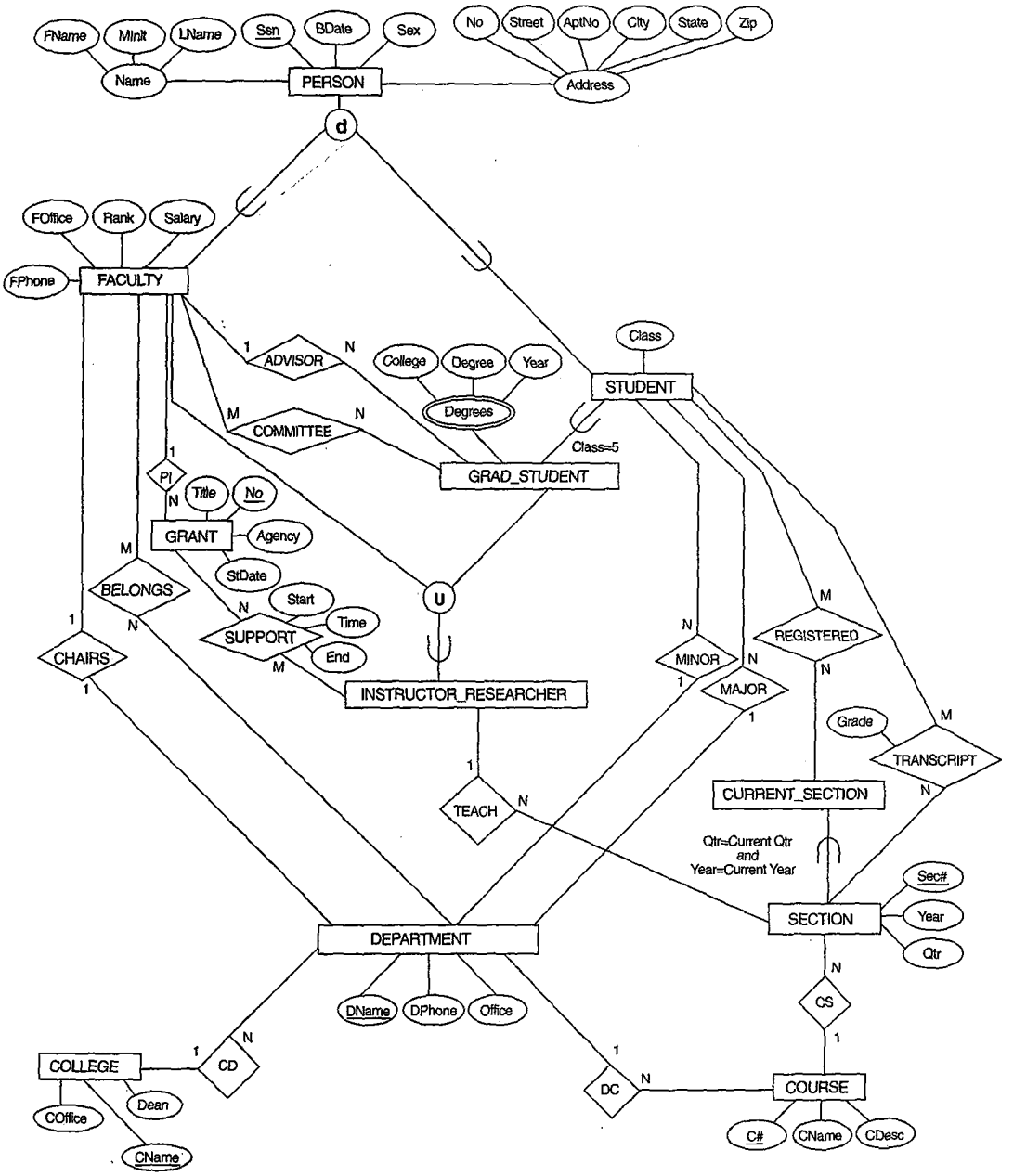


Figure 4.10 An EER conceptual schema for a UNIVERSITY database.

Petitioner Uber Ex-1052, 0114

section is related to the instructor who taught or is teaching it ([TEACH], if that instructor is in the database).

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

Formal Definitions for the EER Model Concepts. We now summarize the EER model concepts and give formal definitions. A **class**[13] is a set or collection of entities; this includes any of the EER schema constructs that group entities such as entity types, subclasses, superclasses, and categories. A **subclass** S is a class whose entities must always be a subset of the entities in another class, called the **superclass** C of the **superclass/subclass** (or **IS-A**) **relationship.** We denote such a relationship by C/S. For such a superclass/subclass relationship, we must always have

$$S \subseteq C$$

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

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

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

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

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

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

13. The use of the word *class* here differs from its more common use in object-oriented programming languages such as C++. In C++, a class is a structured type definition along with its applicable functions (operations).

Petitioner Uber Ex-1052, 0115

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

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

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

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

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

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

4.6 Conceptual Object Modeling Using UML Class Diagrams

Object modeling methodologies, such as UML (Universal Modeling Language) and OMT (Object Modeling Technique) are becoming increasingly popular. Although these methodologies were developed mainly for software design, a major part of software design involves designing the databases that will be accessed by the software modules. Hence, an important part of these methodologies—namely, the *class diagrams*[14]—are similar to EER diagrams in many ways. Unfortunately, the terminology often differs. In this section, we briefly review some of the notation, terminology, and concepts used in UML class diagrams, and compare them with EER terminology and notation. Figure 4.11 shows how the COMPANY ER database schema of Figure 3.15 can be displayed using UML notation. The *entity types* in Figure 3.15 are modeled as *classes* in Figure 4.11. An *entity* in ER corresponds to an *object* in UML.

In UML class diagrams, a class is displayed as a box (see Figure 4.11) that includes three sections: the top section gives the class name; the middle section includes the attributes for individual objects of the class; and the last section includes operations that can be applied to these objects. Operations are *not* specified in EER diagrams. Consider the EMPLOYEE class in Figure 4.11. Its attributes are Name, Ssn, Bdate, Sex, Address, and Salary. The designer can optionally specify the *domain of* an attribute if desired, by placing a : followed by the domain name or description (see the Name, Sex, and Bdate attributes of EMPLOYEE in Figure 4.11). A composite attribute is modeled as a *structured domain*, as illus-

14. A **class** is similar to an *entity type* except that it can have *operations*.

Petitioner Uber Ex-1052, 0116

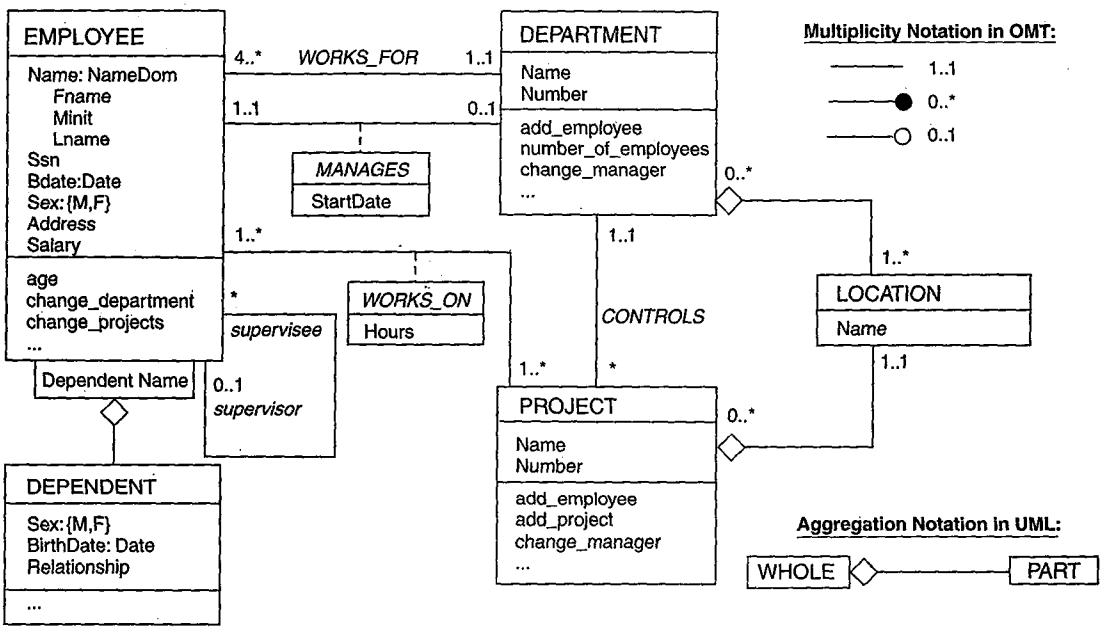


Figure 4.11 The UML conceptual schema for the COMPANY database in Figure 3.15.

trated by the Name attribute of EMPLOYEE. A multivalued attribute will generally be modeled as a separate class, as illustrated by the LOCATION class in Figure 4.11.

Relationship types are called *associations* in UML terminology, and relationship instances are called *links*. A binary association (binary relationship type) is represented as a line connecting the participating classes (entity types), and may (optional) have a name. A relationship attribute, called a *link attribute*, is placed in a box that is connected to the association's line by a dashed line. The (min, max) notation described in Section 3.7.4 is used to specify relationship constraints, which are called *multiplicities* in UML terminology. Multiplicities are specified in the form *min..max*, and an asterisk (*) indicates no maximum limit on participation. However, the multiplicities are placed *on the opposite ends of the relationship* when compared to the notation discussed in Section 3.7.4 (compare Figures 4.11 and 3.15). In UML, a single asterisk indicates a multiplicity of 0..*, and a single 1 indicates a multiplicity of 1..1. A recursive relationship (see Section 3.4.2) is called a *reflexive association* in UML, and the role names—like the multiplicities—are placed at the opposite ends of an association when compared to the placing of role names in Figure 3.15.

In UML, there are two types of relationships: association and aggregation. *Aggregation* is meant to represent a relationship between a whole object and its component parts, and

Petitioner Uber Ex-1052, 0117

it has a distinct diagrammatic notation. In Figure 4.11, we modeled the locations of a department and the single location of a project as aggregations. However, aggregation and association do not have different structural properties, and the choice as to which type of relationship to use is somewhat subjective. In the EER model, both are represented as relationships. UML also distinguishes between *unidirectional* associations/aggregations—which are displayed with an arrow to indicate that only one direction for accessing related objects is needed—and *bi-directional* associations/aggregations—which are the default. In addition, relationship instances may be specified to be *ordered*. Relationship (association) names are *optional* in UML, and relationship attributes are displayed in a box attached with a dashed line to the line representing the association/aggregation (see StartDate and Hours in Figure 4.11).

The operations given in each class are derived from the functional requirements of the application, as we discussed in Section 3.1. It is generally sufficient to specify the operation names initially for the logical operations that are expected to be applied to individual objects of a class, as shown in Figure 4.11. As the design is refined, more details are added, such as the exact argument types (parameters) for each operation, plus a functional description of each operation. UML has *function descriptions* and *sequence diagrams* to specify some of the operation details, but these are beyond the scope of our discussion, and are usually described in software engineering texts.

Weak entities can be modeled using the construct called *qualified association* (or *qualified aggregation*) in UML; this can represent both the identifying relationship and the partial key, which is placed in a box attached to the owner class. This is illustrated by the DEPENDENT class and its qualified aggregation to EMPLOYEE in Figure 4.11.[15]

Figure 4.12 illustrates the UML notation for generalization/specialization by giving a possible UML class diagram corresponding to the EER diagram in Figure 4.7. A blank triangle indicates a *disjoint* specialization/generalization, and a filled triangle indicates *overlapping*.

The above discussion and examples give a brief overview of UML class diagrams and terminology. There are many details that we have not discussed because they are outside the scope of this book. The bibliography at this end of the chapter gives some references to books that describe complete details of UML.

4.7 Relationship Types of a Degree Higher Than Two

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

15. Qualified associations are not restricted to modeling weak entities, and they can be used to model other situations as well.

Petitioner Uber Ex-1052, 0118

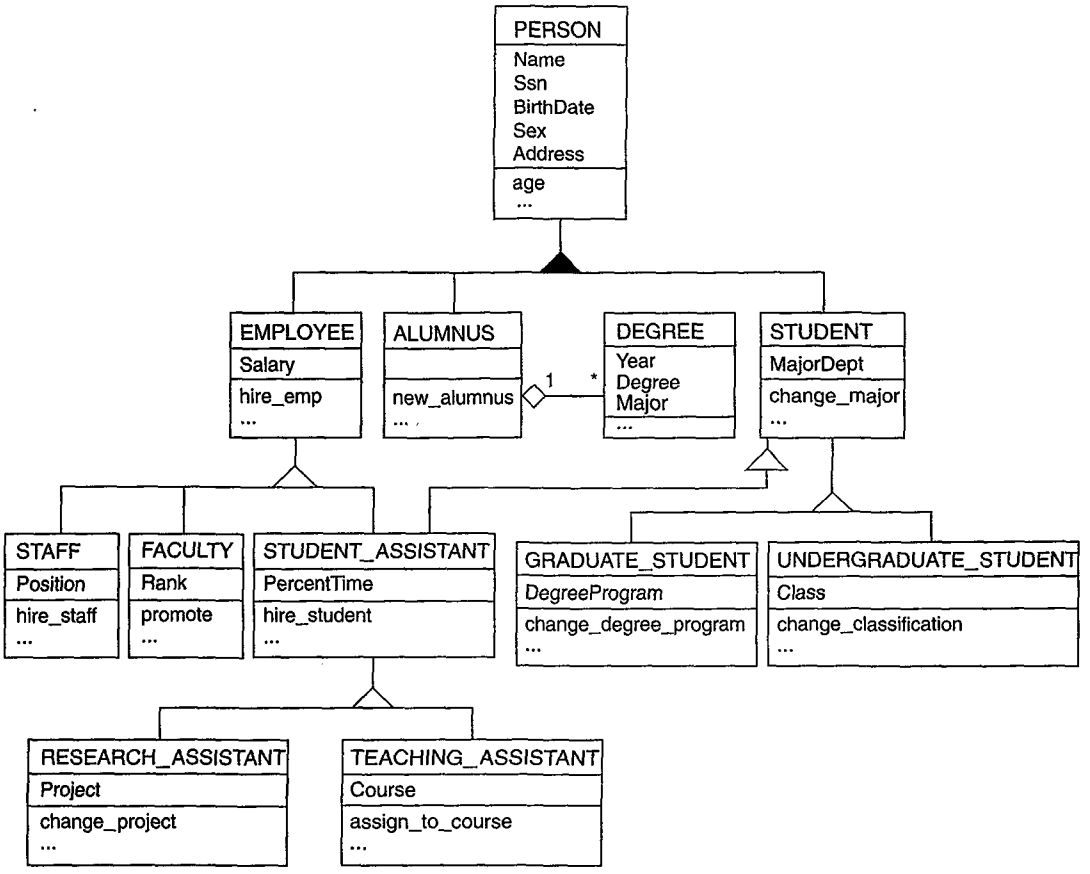


Figure 4.12 Specialization/generalization notation in UML shown by a class diagram corresponding to the EER diagram in Figure 4.7.

Choosing Between Binary and Ternary (or Higher-Degree) Relationships. The ER diagram notation for a ternary relationship type is shown in Figure 4.13(a), which displays the schema for the SUPPLY relationship type that was displayed at the instance level in Figure 3.10. In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

Figure 4.13(b) shows an ER diagram for the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents more information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART,

Petitioner Uber Ex-1052, 0119

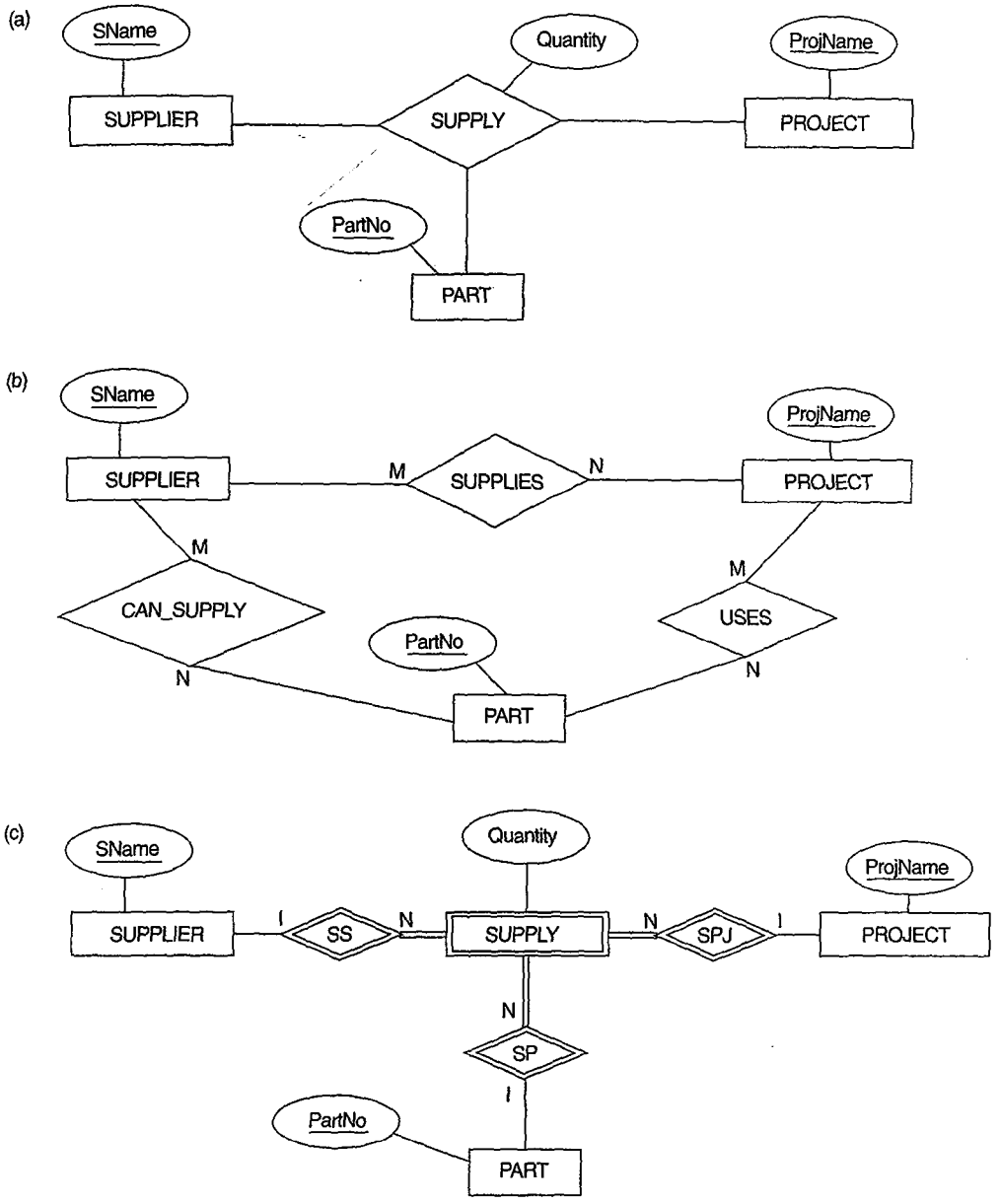


Figure 4.13 An illustration of ternary relationship types. (a) The ternary relationship type SUPPLY. (b) Three binary relationship types that are not equivalent to the ternary relationship type SUPPLY. (c) SUPPLY represented as a weak entity type.

Petitioner Uber Ex-1052, 0120

includes an instance (s, p) whenever supplier s *can supply* part p (to any project); USES, between PROJECT and PART, includes an instance (j, p) whenever project j *uses* part p; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s *supplies some part* to project j. The existence of three relationship instances (s, p), (j, p), and (s, j) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (s, j, p) exists in the ternary relationship SUPPLY because the *meaning is different*! It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree n or should be broken down into several relationship types of smaller degrees. The designer must base this decision on the semantics or meaning of the particular situation being represented. The typical solution is to include the ternary relationship *plus* one or more of the binary relationships, as needed.

Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships. The three participating entity types SUPPLIER, PART, and PROJECT are together the owner entity types (see Figure 4.13c). Hence, an entity in the weak entity type SUPPLY of Figure 4.13(c) is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

Another example is shown in Figure 4.14. The ternary relationship type OFFERS represents information on instructors offering courses during particular semesters; hence it includes a relationship instance (i, s, c) whenever instructor i offers course c during semester s. The three binary relationship types shown in Figure 4.14 have the following meaning: CAN_TEACH relates a course to the instructors who *can teach* that course; TAUGHT_DURING relates a semester to the instructors who *taught some course* during that semester;

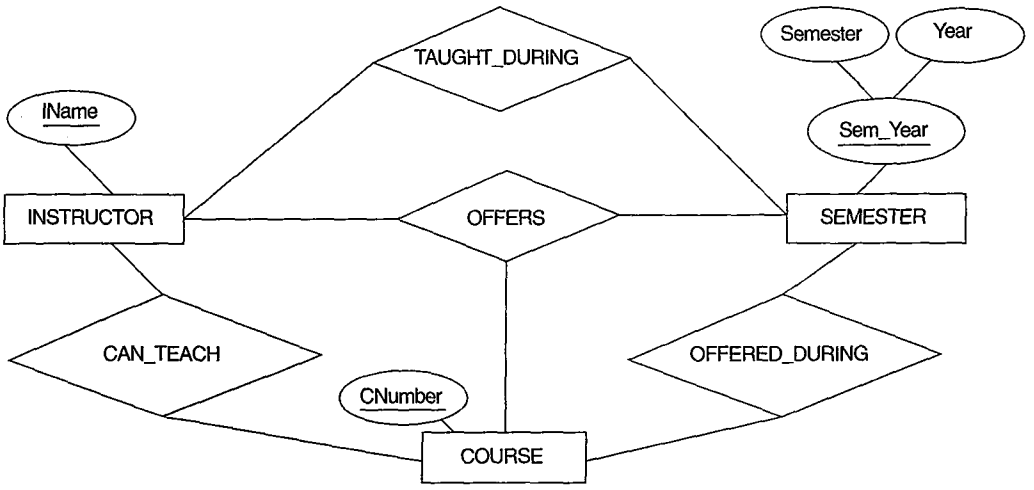


Figure 4.14 Another example of ternary versus binary relationship types.

Petitioner Uber Ex-1052, 0121

and OFFERED_DURING relates a semester to the courses offered during that semester *by any instructor*. In general, these ternary and binary relationships represent different information, but certain constraints should hold among the relationships. For example, a relationship instance (i, s, c) should not exist in OFFERS *unless* an instance (i, s) exists in TAUGHT_DURING, an instance (s, c) exists in OFFERED_DURING, and an instance (i, c) exists in CAN_TEACH. However, the reverse is not always true; we may have instances (i, s), (s, c), and (i, c) in the three binary relationship types with no corresponding instance (i, s, c) in OFFERS. Under certain *additional constraints*, the latter may hold—for example, if the CAN_TEACH relationship is 1:1 (an instructor can teach one course, and a course can be taught by only one instructor). The schema designer must analyze each specific situation to decide which of the binary and ternary relationship types are needed.

Notice that it is possible to have a weak entity type with a ternary (or *n*-ary) identifying relationship type. In this case, the weak entity type can have *several* owner entity types. An example is shown in Figure 4.15.

Constraints on Ternary (or Higher-Degree) Relationships. There are two notations for specifying structural constraints on n-ary relationships, and they specify different constraints. They should thus *both be used* if it is important to fully specify the structural constraints on a ternary or higher-degree relationship. The first notation is based on the cardinality ratio notation of binary relationships, displayed in Figure 3.2. Here, a 1, M, or N is specified on each participation arc. Let us illustrate this constraint using the SUPPLY relationship in Figure 4.13.

Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where s is a SUPPLIER, j is a PROJECT, and p is a PART. Suppose that the constraint exists that for a particular project-part combination, only one supplier will be used (only one supplier supplies a particular part to a particular project). In this case, we place 1 on the SUPPLIER participation, and M, N on the PROJECT, PART participations in Figure 4.13. This

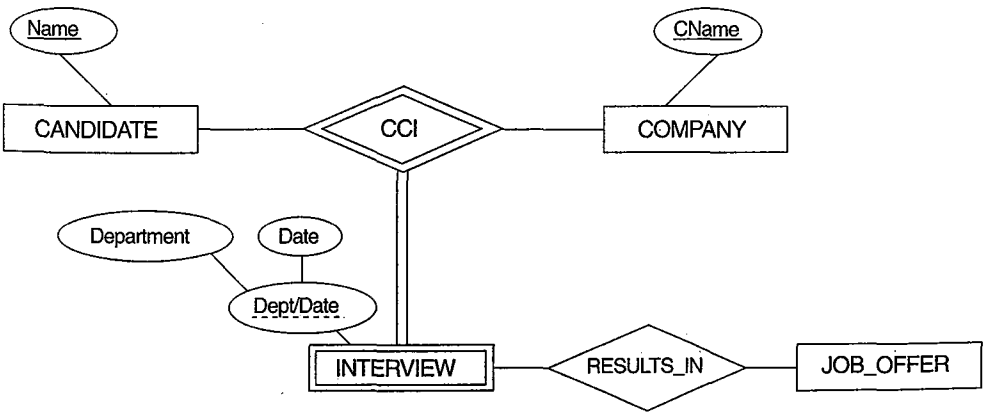


Figure 4.15 A weak entity type INTERVIEW, with a ternary identifying relationship type.

Petitioner Uber Ex-1052, 0122

specifies the constraint that a particular (j, p) combination can appear *at most once* in the relationship set. Hence, any relationship instance (s, j, p) is *uniquely identified* in the relationship set by its (j, p) combination, which makes (j, p) a key for the relationship set. In general, the participations that have a 1 specified on them are not required to be part of the key for the relationship set.[16]

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

4.8 Data Abstraction and Knowledge Representation Concepts

In this section we discuss in abstract terms some of the modeling concepts that we described quite specifically in our presentation of the ER and EER models in Chapters 3 and 4. This terminology is used both in conceptual data modeling and in artificial intelligence literature when discussing **knowledge representation** (abbreviated as KR). The goal of KR techniques is to develop concepts for accurately modeling some **domain of discourse** by creating an **ontology**[18] that describes the concepts of the domain. This is then used to store and manipulate knowledge for drawing inferences, making decisions, or just answering questions. The goals of KR are similar to those of *semantic data models*, but we can summarize some important similarities and differences between the two disciplines:

- Both disciplines use an abstraction process to identify common properties and important aspects of objects in the miniworld (domain of discourse) while suppressing insignificant differences and unimportant details.
- Both disciplines provide concepts, constraints, operations, and languages for defining data and representing knowledge.
- KR is generally broader in scope than semantic data models. Different forms of knowledge, such as rules (used in inference, deduction, and search), incomplete and default knowledge, and temporal and spatial knowledge, are represented in KR schemes. Database models are being expanded to include some of these concepts (see Chapter 23).
- KR schemes include **reasoning mechanisms** that deduce additional facts from the facts stored in a database. Hence, whereas most current database systems are limited to answering direct queries, knowledge-based systems using KR schemes can answer

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

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

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

Petitioner Uber Ex-1052, 0123

queries that involve **inferences** over the stored data. Database technology is being extended with inference mechanisms (see Chapter 25).

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

In this section we discuss four **abstraction concepts** that are used in both semantic data models, such as the EER model, and KR schemes: (1) classification and instantiation, (2) identification, (3) specialization and generalization, and (4) aggregation and association. The paired concepts of classification and instantiation are inverses of one another, as are generalization and specialization. The concepts of aggregation and association are also related. We discuss these abstract concepts and their relation to the concrete representations used in the EER model to clarify the data abstraction process and to improve our understanding of the related process of conceptual schema design.

4.8.1 Classification and Instantiation

The process of **classification** involves systematically assigning similar objects/entities to object classes/entity types. We can now describe (in DB) or reason about (in KR) the classes rather than the individual objects. Collections of objects share the same types of attributes, relationships, and constraints, and by classifying objects we simplify the process of discovering their properties. **Instantiation** is the inverse of classification and refers to the generation and specific examination of distinct objects of a class. Hence, an object instance is related to its object class by the IS-AN-INSTANCE-OF relationship.[19]

In general, the objects of a class should have a similar type structure. However, some objects may display properties that differ in some respects from the other objects of the class; these **exception objects** also need to be modeled, and KR schemes allow more varied exceptions than do database models. In addition, certain properties apply to the class as a whole and not to the individual objects; KR schemes allow such **class properties.**[20]

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

19. UML diagrams allow a form of instantiation by permitting the display of individual objects. We *did not* describe this feature in Section 4.6.

20. UML diagrams also allow specification of class properties.

Petitioner Uber Ex-1052, 0124

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

4.8.2 Identification

Identification is the abstraction process whereby classes and objects are made uniquely identifiable by means of some **identifier.** For example, a class name uniquely identifies a whole class. An additional mechanism is necessary for telling distinct object instances apart by means of object identifiers. Moreover, it is necessary to identify multiple manifestations in the database of the same real-world object. For example, we may have a tuple <*Matthew Clarke*, 610618, 376-9821> in a PERSON relation and another tuple <*301-54-0836*, CS, 3.8> in a STUDENT relation that happens to represent the same real-world entity. There is no way to identify the fact that these two database objects (tuples) represent the same real-world entity unless we make a provision *at design time* for appropriate cross-referencing to supply this identification. Hence, identification is needed at two levels:

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

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

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

4.8.3 Specialization and Generalization

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

Petitioner Uber Ex-1052, 0125

4.8.4 Aggregation and Association

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

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

In order to understand the different uses of aggregation better, consider the ER schema shown in Figure 4.16(a), which stores information about interviews by job applicants to various companies. The class COMPANY is an aggregation of the attributes (or component objects) CName (company name) and CAddress (company address), whereas JOB_APPLICANT is an aggregate of Ssn, Name, Address, and Phone. The relationship attributes ContactName and ContactPhone represent the name and phone number of the person in the company who is responsible for the interview. Suppose that some interviews result in job offers, while others do not. We would like to treat INTERVIEW as a class to associate it with JOB_OFFER. The schema shown in Figure 4.16(b) is *incorrect* because it requires each interview relationship instance to have a job offer. The schema shown in Figure 4.16(c) is not allowed, because the ER model does not allow relationships among relationships (although UML does).

One way to represent this situation is to create a higher-level aggregate class composed of COMPANY, JOB_APPLICANT, and INTERVIEW and to relate this class to JOB_OFFER, as shown in Figure 4.16(d). Although the EER model as described in this book does not have this facility, some semantic data models do allow it and call the resulting object a **composite** or **molecular object**. Other models treat entity types and relationship types uniformly and hence permit relationships among relationships (Figure 4.16c).

To represent this situation correctly in the ER model as described here, we need to create a new weak entity type INTERVIEW, as shown in Figure 4.16(e), and relate it to JOB_OFFER. Hence, we can always represent these situations correctly in the ER model by creating additional entity types, although it may be conceptually more desirable to allow direct representation of aggregation as in Figure 4.16(d) or to allow relationships among relationships as in Figure 4.16(c).

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

Petitioner Uber Ex-1052, 0126

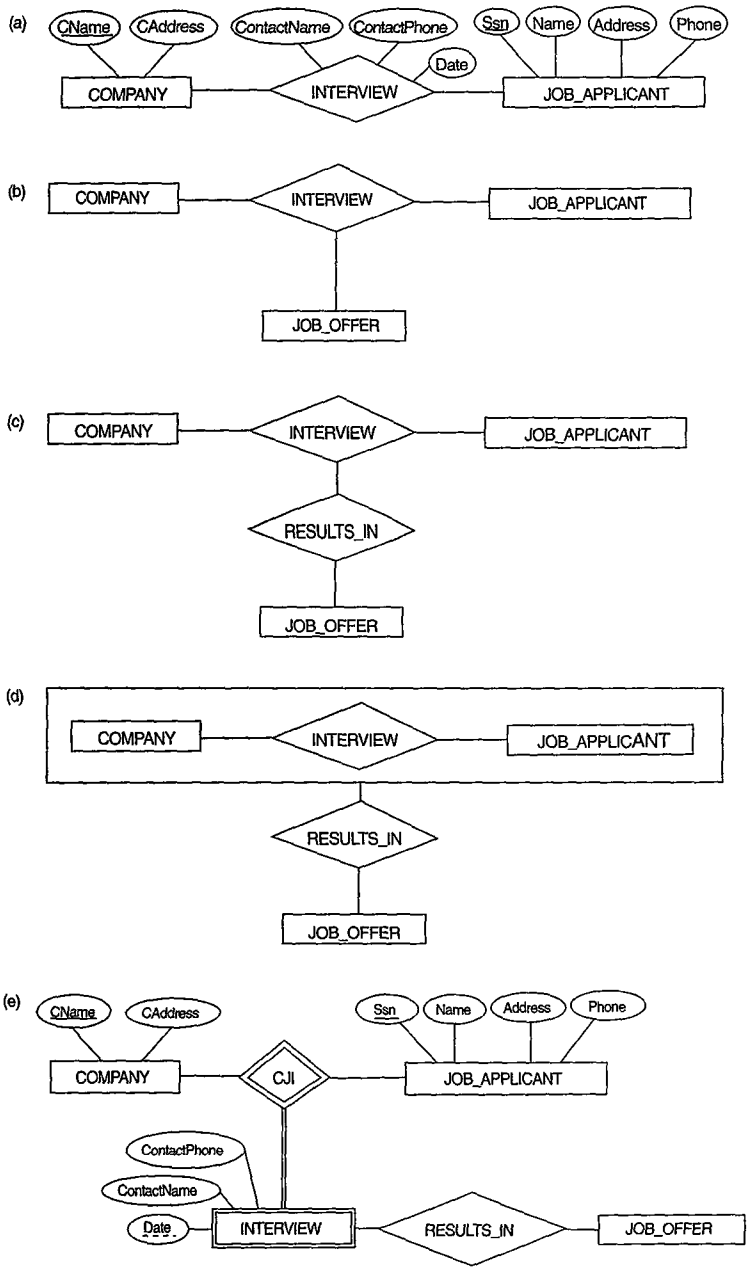


Figure 4.16 An illustration of aggregation. (a) The INTERVIEW relationship type. (b) Including JOB_OFFER in a ternary relationship type (incorrect). (c) Including JOB_OFFER by having a relationship in which another relationship participates (generally not allowed in ER). (d) Using aggregation and a composite (molecular) object (generally not allowed in ER). (e) Correct representation in ER.

Petitioner Uber Ex-1052, 0127

4.9 Summary

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

We then showed how to display these new constructs in an EER diagram. We also discussed the various types of constraints that may apply to specialization or generalization. The two main constraints are total/partial and disjoint/overlapping. In addition, a defining predicate for a subclass or a defining attribute for a specialization may be specified. We discussed the differences between user-defined and predicate-defined subclasses and between user-defined and attribute-defined specializations. Finally, we discussed the concept of a category, which is a subset of the union of two or more classes, and we gave formal definitions of all the concepts presented.

We then introduced the notation and terminology of the Universal Modeling Language (UML), which is being used increasingly in software engineering. We briefly discussed similarities and differences between the UML and EER concepts, notation, and terminology. We also discussed some of the issues concerning the difference between binary and higher-degree relationships, under which circumstances each should be used when designing a conceptual schema, and how different types of constraints on n-ary relationships may be specified. In Section 4.8 we discussed briefly the discipline of knowledge representation and how it is related to semantic data modeling. We also gave an overview and summary of the types of abstract data representation concepts: classification and instantiation, identification, specialization and generalization, aggregation and association. We saw how EER and UML concepts are related to each of these.

Review Questions

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

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

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

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

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

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

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

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

Petitioner Uber Ex-1052, 0128

4.9. How does a category differ from a regular shared subclass? What is a category used for? Illustrate your answer with examples.
4.10. For each of the following UML terms, discuss the corresponding term in the EER model, if any: *object*, *class*, *association*, *aggregation*, *generalization*, *multiplicity*, *attributes*, *discriminator*, *link*, *link attribute*, *reflexive association*, *qualified association*.
4.11. Discuss the main differences between the notation for EER schema diagrams and UML class diagrams by comparing how common concepts are represented in each.
4.12. Discuss the two notations for specifying constraints on n-ary relationships, and what each can be used for.
4.13. List the various data abstraction concepts and the corresponding modeling concepts in the EER model.
4.14. What aggregation feature is missing from the EER model? How can the EER model be further enhanced to support it?
4.15. What are the main similarities and differences between conceptual database modeling techniques and knowledge representation techniques.

Exercises

4.16. Design an EER schema for a database application that you are interested in. Specify all constraints that should hold on the database. Make sure that the schema has at least five entity types, four relationship types, a weak entity type, a superclass/subclass relationship, a category, and an n-ary ($n > 2$) relationship type.
4.17. Consider the BANK ER schema of Figure 3.17, and suppose that it is necessary to keep track of different types of ACCOUNTS (SAVINGS_ACCTS, CHECKING_ACCTS, . . .) and LOANS (CAR_LOANS, HOME_LOANS, . . .). Suppose that it is also desirable to keep track of each account's TRANSACTIONS (deposits, withdrawals, checks, . . .) and each loan's PAYMENTS; both of these include the amount, date, and time. Modify the BANK schema, using ER and EER concepts of specialization and generalization. State any assumptions you make about the additional requirements.
4.18. The following narrative describes a simplified version of the organization of Olympic facilities planned for the 1996 Olympics in Atlanta. Draw an EER diagram that shows the entity types, attributes, relationships, and specializations for this application. State any assumptions you make. The Olympic facilities are divided into sports complexes. Sports complexes are divided into *one-sport* and *multisport* types. Multisport complexes have areas of the complex designated to each sport with a location indicator (e.g., center, NE-corner, etc.). A complex has a location, chief organizing individual, total occupied area, and so on. Each complex holds a series of events (e.g., the track stadium may hold many different races). For each event there is a planned date, duration, number of participants, number of officials, and so on. A roster of all officials will be maintained together with the list of events each official will be involved in. Different equipment is needed for the events (e.g., goal posts, poles, parallel bars) as well as for maintenance. The two types of facilities (*one-sport* and *multisport*) will have different

Petitioner Uber Ex-1052, 0129

types of information. For each type, the number of facilities needed is kept, together with an approximate budget.

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

Case Study: The Georgia Tech Library (GTL) has approximately 16,000 members, 100,000 titles, and 250,000 volumes (or an average of 2.5 copies per book). About 10 percent of the volumes are out on loan at any one time. The librarians ensure that the books that members want to borrow are available when the members want to borrow them. Also, the librarians must know how many copies of each book are in the library or out on loan at any given time. A catalog of books is available on-line that lists books by author, title, and subject area. For each title in the library, a book description is kept in the catalog that ranges from one sentence to several pages. The reference librarians want to be able to access this description when members request information about a book. Library staff is divided into chief librarian, departmental associate librarians, reference librarians, check-out staff, and library assistants. Books can be checked out for 21 days. Members are allowed to have only five books out at a time. Members usually return books within three to four weeks. Most members know that they have one week of grace before a notice is sent to them, so they try to get the book returned before the grace period ends. About 5 percent of the members have to be sent reminders to return a book. Most overdue books are returned within a month of the due date. Approximately 5 percent of the overdue books are either kept or never returned. The most active members of the library are defined as those who borrow at least ten times during the year. The top 1 percent of membership does 15 percent of the borrowing, and the top 10 percent of the membership does 40 percent of the borrowing. About 20 percent of the members are totally inactive in that they are members but do never borrow. To become a member of the library, applicants fill out a form including their SSN, campus and home mailing addresses, and phone numbers. The librarians then issue a numbered, machine-readable card with the member's photo on it. This card is good for four years. A month before a card expires, a notice is sent to a member for renewal. Professors at the institute are considered automatic members. When a new faculty member joins the institute, his or her information is pulled from the employee records and a library card is mailed to his or her campus address. Professors are allowed to check out books for three-month intervals and have a two-week grace period. Renewal notices to professors are sent to the campus address. The library does not lend some books, such as reference books, rare books, and maps. The librarians must differentiate between books that can be lent and those that cannot be lent. In addition, the librarians have a list of some books they are interested in acquiring but cannot obtain, such as rare or out-of-print books and books that were lost

Petitioner Uber Ex-1052, 0130

or destroyed but have not been replaced. The librarians must have a system that keeps track of books that cannot be lent as well as books that they are interested in acquiring. Some books may have the same title; therefore, the title cannot be used as a means of identification. Every book is identified by its International Standard Book Number (ISBN), a unique international code assigned to all books. Two books with the same title can have different ISBNs if they are in different languages or have different bindings (hard cover or soft cover). Editions of the same book have different ISBNs. The proposed database system must be designed to keep track of the members, the books, the catalog, and the borrowing activity.

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

- The museum has a collection of ART_OBJECTs. Each ART_OBJECT has a unique IdNo, an Artist (if known), a Year (when it was created, if known), a Title, and a Description. The art objects are categorized in several ways as discussed below.
- ART_OBJECTs are categorized based on their type. There are three main types: PAINTING, SCULPTURE, and STATUE, plus another type called OTHER to accommodate objects that do not fall into one of the three main types.
- A PAINTING has a PaintType (oil, watercolor, etc.), material on which it is DrawnOn (paper, canvas, wood, etc.), and Style (modern, abstract, etc.).
- A SCULPTURE has a Material from which it was created (wood, stone, etc.), Height, Weight, and Style.
- An art object in the OTHER category has a Type (print, photo, etc.) and Style.
- ART_OBJECTs are also categorized as PERMANENT_COLLECTION that are owned by the museum (which has information on the DateAcquired, whether it is OnDisplay or stored, and Cost) or BORROWED, which has information on the Collection (from which it was borrowed), DateBorrowed, and DateReturned.
- ART_OBJECTs also have information describing their country/culture using information on country/culture of Origin (Italian, Egyptian, American, Indian, etc.), Epoch (Renaissance, Modern, Ancient, etc.).
- The museum keeps track of ARTIST's information, if known: Name, DateBorn, DateDied (if not living), CountryOfOrigin, Epoch, MainStyle, Description. The Name is assumed to be unique.
- Different EXHIBITIONs occur, each having a Name, StartDate, EndDate, and is related to all the art objects that were on display during the exhibition.
- Information is kept on other COLLECTIONs with which the museum interacts, including Name (unique), Type (museum, personal, etc.), Description, Address, Phone, and current ContactPerson.

Draw an EER schema diagram for this application. Discuss any assumptions you made, and that justify your EER design choices.

Petitioner Uber Ex-1052, 0131

4.21. Figure 4.17 shows an example of an EER diagram for a small private airport database that is used to keep track of airplanes, their owners, airport employees, and pilots. From the requirements for this database, the following information was collected. Each airplane has a registration number [Reg#], is of a particular plane type [OF-TYPE], and is stored in a particular hangar [STORED-IN]. Each plane type has a model number [Model], a capacity [Capacity], and a weight [Weight]. Each hangar has a number [Number], a capacity [Capacity], and a location [Location]. The

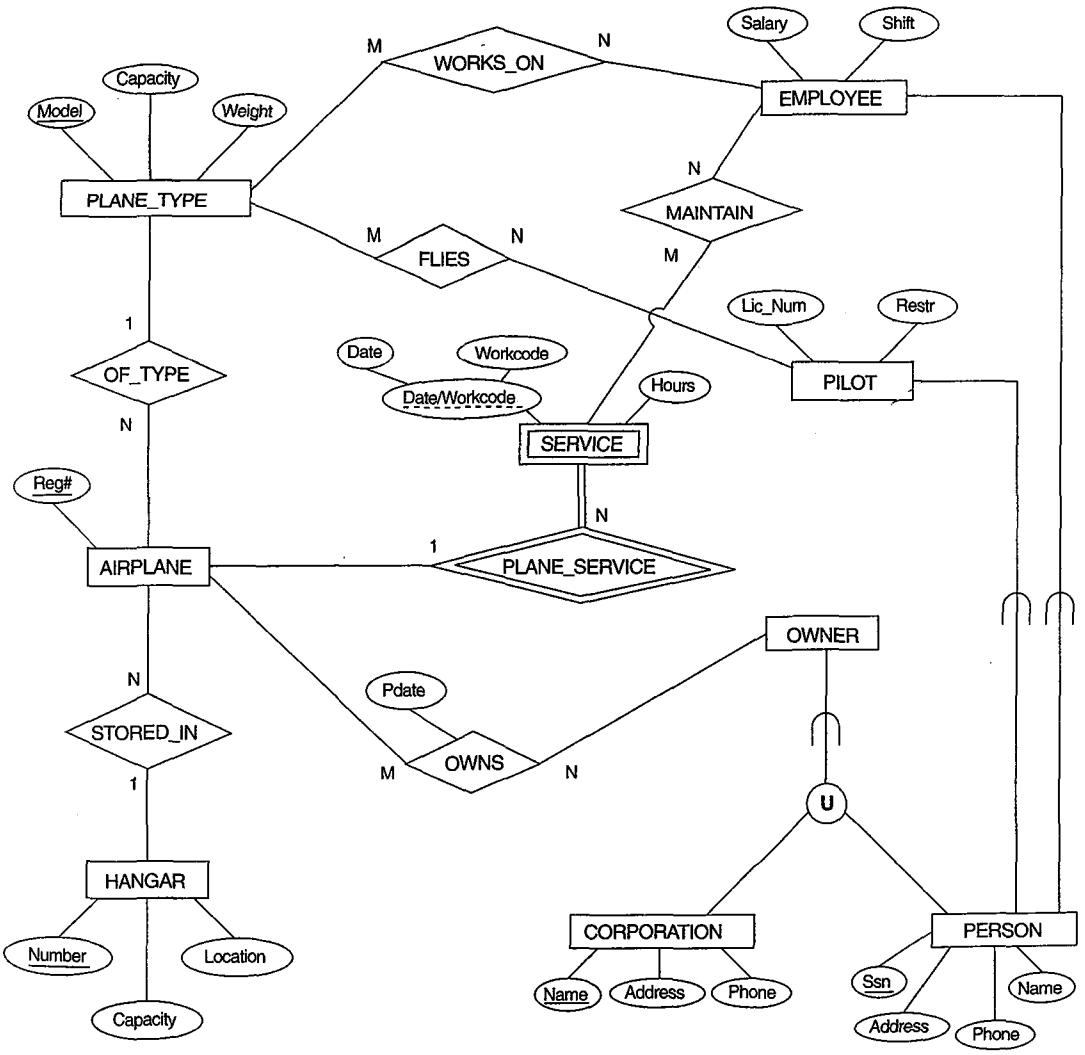


Figure 4.17 EER schema for a SMALL AIRPORT database.

Petitioner Uber Ex-1052, 0132

database also keeps track of the owners of each plane [OWNS] and the employees who have maintained the plane [MAINTAIN]. Each relationship instance in OWNS relates an airplane to an owner and includes the purchase date [Pdate]. Each relationship instance in MAINTAIN relates an employee to a service record [SERVICE]. Each plane undergoes service many times; hence, it is related by [PLANE-SERVICE] to a number of service records. A service record includes as attributes the date of maintenance [Date], the number of hours spent on the work [Hours], and the type of work done [Workcode]. We use a weak entity type [SERVICE] to represent airplane service, because the airplane registration number is used to identify a service record. An owner is either a person or a corporation. Hence, we use a union category [OWNER] that is a subset of the union of corporation [CORPORATION] and person [PERSON] entity types. Both pilots [PILOT] and employees [EMPLOYEE] are subclasses of PERSON. Each pilot has specific attributes license number [Lic-Num] and restrictions [Restr]; each employee has specific attributes salary [Salary] and shift worked [Shift]. All person entities in the database have data kept on their social security number [Ssn], name [Name], address [Address], and telephone number [Phone]. For corporation entities, the data kept includes name [Name], address [Address], and telephone number [Phone]. The database also keeps track of the types of planes each pilot is authorized to fly [FLIES] and the types of planes each employee can do maintenance work on [WORKS-ON]. Show how the SMALL AIRPORT EER schema of Figure 4.17 may be represented in UML notation. (*Note:* We have not discussed how to represent categories (union types) in UML so you do not have to map the categories in this and the following question).

4.22. Show how the UNIVERSITY EER schema of Figure 4.10 may be represented in UML notation.

Selected Bibliography

Many papers have proposed conceptual or semantic data models. We give a representative list here. One group of papers, including Abrial (1974), Senko's DIAM model (1975), the NIAM method (Verheijen and VanBekkum 1982), and Bracchi et al. (1976), presents semantic models that are based on the concept of binary relationships. Another group of early papers discusses methods for extending the relational model to enhance its modeling capabilities. This includes the papers by Schmid and Swenson (1975), Navathe and Schkolnick (1978), Codd's RM/T model (1979), Furtado (1978), and the structural model of Wiederhold and Elmasri (1979).

The ER model was proposed originally by Chen (1976) and is formalized in Ng (1981). Since then, numerous extensions of its modeling capabilities have been proposed, as in Scheuermann et al. (1979), Dos Santos et al. (1979), Teorey et al. (1986), Gogolla and Hohenstein (1991), and the Entity-Category-Relationship (ECR) model of Elmasri et al. (1985). Smith and Smith (1977) present the concepts of generalization and aggregation. The semantic data model of Hammer and McLeod (1981) introduced the concepts of class/subclass lattices, as well as other advanced modeling concepts.

Petitioner Uber Ex-1052, 0133

A survey of semantic data modeling appears in Hull and King (1987). Another survey of conceptual modeling is Pillalamarri et al. (1988). Eick (1991) discusses design and transformations of conceptual schemas. Analysis of constraints for *n*-ary relationships is given in Soutou (1998). UML is described in detail in Booch, Rumbaugh, and Jacobson (1999).

Petitioner Uber Ex-1052, 0134

CHAPTER 5 Record Storage and Primary File Organizations

Databases are stored physically as files of records, which are typically stored on magnetic disks. This chapter and the next deal with the organization of databases in storage and the techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called indexes. We start in Section 5.1 by introducing the concepts of computer storage hierarchies and how they are used in database systems. Section 5.2 is devoted to a description of magnetic disk storage devices and their characteristics, and we also briefly describe magnetic tape storage devices. Section 5.3 describes a more recent data storage system alternative called RAID (Redundant Arrays of Inexpensive (or Independent) Disks), which provides better reliability and improved performance. Having discussed different storage technologies, we then turn our attention to the methods for organizing data on disks. Section 5.4 covers the technique of double buffering, which is used to speed retrieval of multiple disk blocks. In Section 5.5 we discuss various ways of formatting and storing records of a file on disk. Section 5.6 discusses the various types of operations that are typically applied to records of a file. We then present three primary methods for organizing records of a file on disk: unordered records, discussed in Section 5.7; ordered records, in Section 5.8; and hashed records, in Section 5.9.

Section 5.10 very briefly discusses files of mixed records and other primary methods for organizing records, such as B-trees. These are particularly relevant for storage of object-oriented databases, which we discuss later in Chapters 11 and 12. In Chapter 6 we discuss techniques for creating auxiliary data structures, called indexes, that speed up the

Petitioner Uber Ex-1052, 0135

search for and retrieval of records. These techniques involve storage of auxiliary data, called index files, in addition to the file records themselves.

Chapters 5 and 6 may be browsed through or even omitted by readers who have already studied file organizations. They can also be postponed and read later after going through the material on the relational model and the object-oriented models. The material covered here is necessary for understanding some of the later chapters in the book—in particular, Chapters 16 and 18.

5.1 Introduction

The collection of data that makes up a computerized database must be stored physically on some computer **storage medium.** The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a *storage hierarchy* that includes two main categories:

- **Primary storage.** This category includes storage media that can be operated on directly by the computer *central processing unit* (CPU), such as the computer main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity.
- **Secondary storage.** This category includes magnetic disks, optical disks, and tapes. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary storage cannot be processed directly by the CPU; it must first be copied into primary storage.

We will first give an overview of the various storage devices used for primary and secondary storage in Section 5.1.1 and will then discuss how databases are typically handled in the storage hierarchy in Section 5.1.2.

5.1.1 Memory Hierarchies and Storage Devices

In a modern computer system data resides and is transported throughout a hierarchy of storage media. The highest-speed memory is the most expensive and is therefore available with the least capacity. The lowest-speed memory is tape storage, which is essentially available in indefinite storage capacity.

At the *primary storage level,* the memory hierarchy includes at the most expensive end **cache memory,** which is a static RAM (Random Access Memory). Cache memory is typically used by the CPU to speed up execution of programs. The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping programs and data and is popularly called **main memory.** The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility[1] and lower speed compared with static RAM. At the *secondary storage level,* the hierarchy includes magnetic

1. Volatile memory typically loses its contents in case of a power outage, whereas nonvolatile memory does not.

Petitioner Uber Ex-1052, 0136

disks, as well as **mass storage** in the form of CD-ROM (Compact Disk–Read-Only Memory) devices, and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1000 bytes), megabytes (Mbyte or 1 million bytes), gigabytes (Gbyte or 1 billion bytes), and even terabytes (1000 Gbytes).

Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, and portions of the database are read into and written from buffers in main memory as needed. Now that personal computers and workstations have tens of megabytes of data in DRAM, it is becoming possible to load a large fraction of the database into main memory. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), leading to **main memory databases;** these are particularly useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.

Between DRAM and magnetic disk storage, another form of memory, **flash memory,** is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over at a time.[2]

CD-ROM disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten. WORM (Write-Once-Read-Many) disks are a form of optical storage used for archiving data; they allow data to be written once and read any number of times without the possibility of erasing. They hold about half a gigabyte of data per disk and last much longer than magnetic disks. **Optical juke box memories** use an array of CD-ROM platters, which are loaded onto drives on demand. Although optical juke boxes have capacities in the hundreds of gigabytes, their retrieval times are in the hundreds of milliseconds, quite a bit slower than magnetic disks.[3] This type of storage has not become as popular as it was expected to be because of the rapid decrease in cost and increase in capacities of magnetic disks. The DVD (Digital Video Disk) is a recent standard for optical disks allowing four to fifteen gigabytes of storage per disk.

Finally, **magnetic tapes** are used for archiving and backup storage of data. **Tape juke-boxes**—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as **tertiary storage** to hold terabytes of data. For example, NASA's EOS (Earth Observation Satellite) system stores archived databases in this fashion.

It is anticipated that many large organizations will find it normal to have terabyte-sized databases in a few years. The term **very large database** cannot be defined precisely any more because disk storage capacities are on the rise and costs are declining. It may very soon be reserved for databases containing tens of terabytes.

---

2. For example, the INTEL DD28F032SA is a 32-megabit capacity flash memory with 70-nanosecond access speed, and 430 KB/second write transfer rate.

3. Their rotational speeds are lower (around 400 rpm), giving higher latency delays and low transfer rates (around 100 to 200 KB per second).

Petitioner Uber Ex-1052, 0137

5.1.2 Storage of Databases

Databases typically store large amounts of data that must *persist* over long periods of time. The data is accessed and processed repeatedly during this period. This contrasts with the notion of *transient* data structures that persist for only a limited time during program execution. Most databases are stored permanently (or *persistently*) on magnetic disk secondary storage, for the following reasons:

- Generally, databases are too large to fit entirely in main memory.
- The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as **nonvolatile storage,** whereas main memory is often called **volatile storage.**
- The cost of storage per unit of data is an order of magnitude less for disk than for primary storage.

Some of the newer technologies—such as optical disks, DVDs, and tape jukeboxes—are likely to provide viable alternatives to the use of magnetic disks. Databases in the future may therefore reside at different levels of the memory hierarchy from those described in Section 5.1.1. For now, however, it is important to study and understand the properties and characteristics of magnetic disks and the way data files can be organized on disk in order to design effective databases with acceptable performance.

Magnetic tapes are frequently used as a storage medium for backing up the database because storage on tape costs even less than storage on disk. However, access to data on tape is quite slow. Data stored on tapes is **off-line;** that is, some intervention by an operator—or an automatic loading device—to load a tape is needed before this data becomes available. In contrast, disks are **on-line** devices that can be accessed directly at any time.

The techniques used to store large amounts of structured data on disk are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific DBMS. Usually, the DBMS has several options available for organizing the data, and the process of **physical database design** involves choosing from among the options the particular data organization techniques that best suit the given application requirements. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as **files** of **records.** Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently whenever they are needed.

There are several **primary file organizations,** which determine how the records of a file are *physically placed* on the disk, *and hence how the records can be accessed.* A *heap file* (or

Petitioner Uber Ex-1052, 0138

unordered file) places the records on disk in no particular order by appending new records at the end of the file, whereas a *sorted file* (or *sequential file*) keeps the records ordered by the value of a particular field (called the sort key). A *hashed file* uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk. Other primary file organizations, such as *B-trees*, use tree structures. We discuss primary file organizations in Sections 5.7 through 5.10. A **secondary organization** or **auxiliary access structure** allows efficient access to the records of a file based on *alternate fields* than those that have been used for the primary file organization. Most of these exist as indexes and will be discussed in Chapter 6.

5.2 Secondary Storage Devices

In this section we describe some characteristics of magnetic disk and magnetic tape storage devices. Readers who have studied these devices already may just browse through this section.

5.2.1 Hardware Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The most basic unit of data on the disk is a single **bit** of information. By magnetizing an area on disk in certain ways, one can make it represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into **bytes** (or **characters**). Byte sizes are typically 4 to 8 bits, depending on the computer and the device. We assume that one character is stored in a single byte, and we use the terms *byte* and *character* interchangeably. The **capacity** of a disk is the number of bytes it can store, which is usually very large. Small floppy disks used with microcomputers typically hold from 400 Kbytes to 1.5 Mbytes; hard disks for micros typically hold from several hundred Mbytes up to a few Gbytes; and large disk packs used with minicomputers and mainframes have capacities that range up to a few tens or hundreds of Gbytes. Disk capacities continue to grow as technology improves.

Whatever their capacity, disks are all made of magnetic material shaped as a thin circular disk (Figure 5.1a) and protected by a plastic or acrylic cover. A disk is **single-sided** if it stores information on only one of its surfaces and **double-sided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack** (Figure 5.1b), which may include many disks and hence many surfaces. Information is stored on a disk surface in concentric circles of *small width*,[4] each having a distinct diameter. Each circle is called a **track.** For disk packs, the tracks with the same diameter on the various surfaces are called a **cylinder** because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on one cylinder can be retrieved much faster than if it were distributed among different cylinders.

The number of tracks on a disk ranges from a few hundred to a few thousand, and the capacity of each track typically ranges from tens of Kbytes to 150 Kbytes. Because a track usu-

4. In some disks, the circles are now connected into a kind of continuous spiral.

Petitioner Uber Ex-1052, 0139

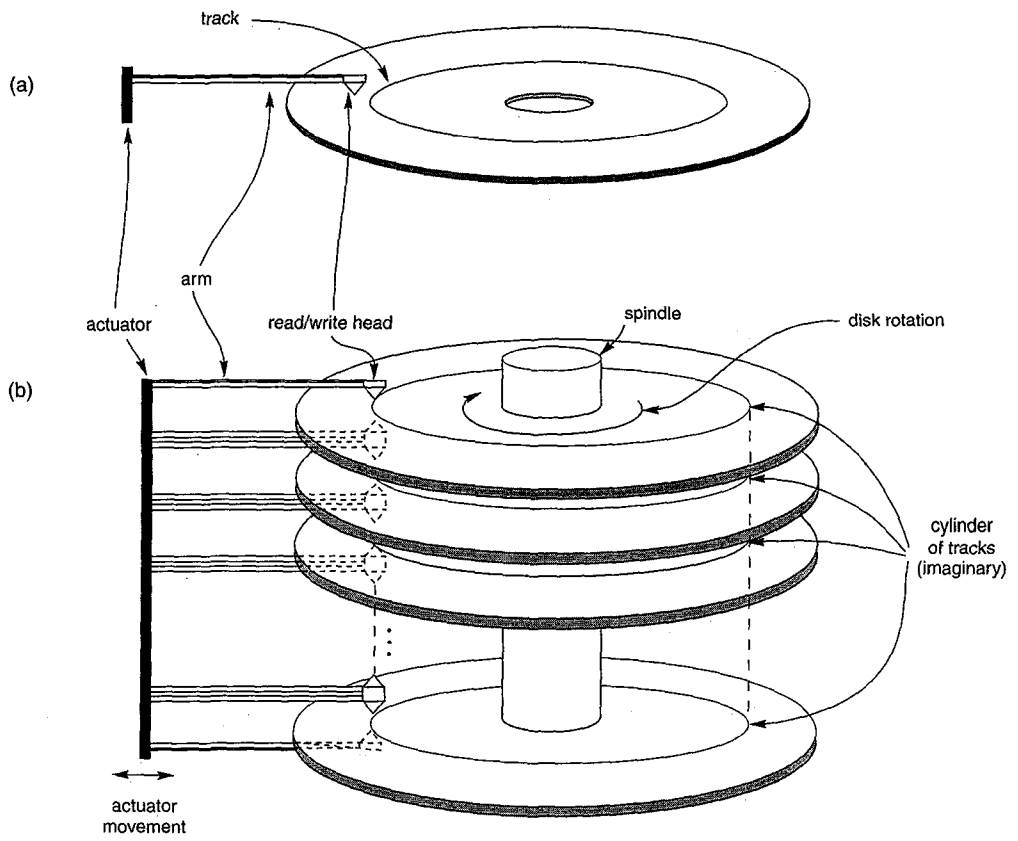


Figure 5.1 (a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.

ally contains a large amount of information, it is divided into smaller blocks or sectors. The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed. One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector (Figure 5.2a). Several other sector organizations are possible, one of which is to have the sectors subtend smaller angles at the center as one moves away, thus maintaining a uniform density of recording (Figure 5.2b). Not all disks have their tracks divided into sectors.

The division of a track into equal-sized **disk blocks** (or **pages**) is set by the operating system during disk **formatting** (or **initialization**). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range from 512 to 4096 bytes. A disk with hard-coded sectors often has the sectors subdivided into blocks during initialization. Blocks are separated by fixed-size **interblock gaps,** which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each interblock gap. Table 5.1 represents specifications of a typical disk.

Petitioner Uber Ex-1052, 0140

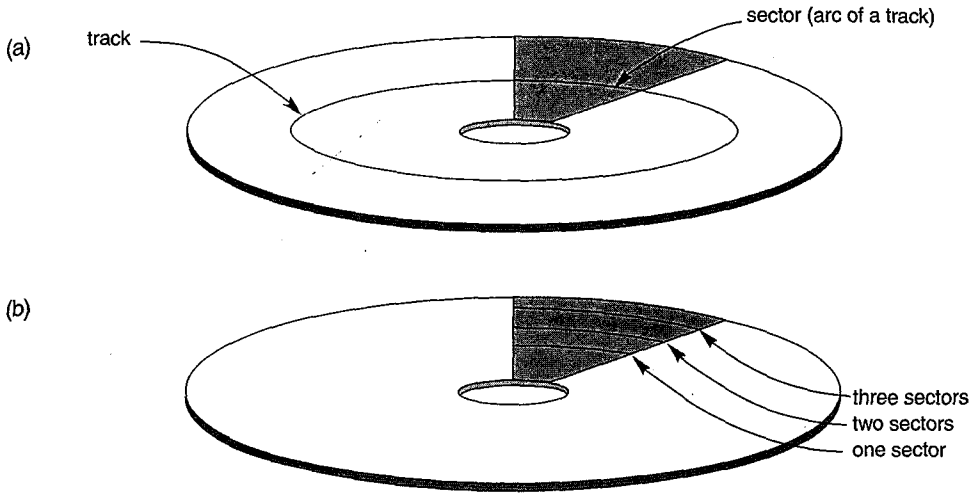


Figure 5.2 Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

There is a continuous improvement in the storage capacity and transfer rates associated with disks; they are also progressively getting cheaper—currently costing only a fraction of a dollar per megabyte of disk storage. Costs are going down so rapidly that costs as low as one cent per megabyte or $10K per terabyte by the year 2001 are being forecast.

A disk is a *random access* addressable device. Transfer of data between main memory and disk takes place in units of disk blocks. The **hardware address** of a block—a combination of a surface number, track number (within the surface), and block number (within the track)—is supplied to the disk input/output (I/O) hardware. The address of a **buffer**—a contiguous reserved area in main storage that holds one block—is also provided. For a **read** command, the block from disk is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a **cluster,** may be transferred as a unit. In this case the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head,** which is part of a system called a **disk drive.** A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm.** Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface (see Figure 5.1b). All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.

Disk drives for hard disks rotate the disk pack continuously at a constant speed (typically ranging between 3600 and 7200 rpm). For a floppy disk, the disk drive begins to rotate the disk whenever a particular read or write request is initiated and ceases rotation

Petitioner Uber Ex-1052, 0141

Table 5.1 Specification of Typical High-end Cheetah Disks from Seagate

Description		
Model number	ST136403LC	ST318203LC
Model name	Cheetah 36	Cheetah 18LP
Form Factor (width)	3.5-inch	3.5-inch
Weight	1.04 Kg	0.59 Kg
Capacity/Interface		
Formatted capacity	36.4 Gbytes, formatted	18.2 Gbytes, formatted
Interface type	80-pin Ultra-2 SCSI	80-pin Ultra-2 SCSI
Configuration		
Number of Discs (physical)	12	6
Number of heads (physical)	24	12
Total cylinders (SCSI only)	9,772	9,801
Total tracks (SCSI only)	N/A	117,612
Bytes per sector	512	512
Track Density (TPI)	N/A tracks/inch	12,580 tracks/inch
Recording Density (BPI, max)	N/A bits/inch	258,048 bits/inch
Performance		
Transfer Rates		
Internal Transfer Rate (min)	193 Mbits/sec	193 Mbits/sec
Internal Transfer Rate (max)	308 Mbits/sec	308 Mbits/sec
Formatted Int transfer rate (min)	18 Mbits/sec	18 Mbits/sec
Formatted Int transfer rate (max)	28 Mbits/sec	28 Mbits/sec
External (I/O) Transfer Rate (max)	80 Mbits/sec	80 Mbits/sec
Seek Times		
Average seek time, read	5.7 msec typical	5.2 msec typical
Average seek time, write	6.5 msec typical	6 msec typical
Track-to-track seek, read	0.6 msec typical	0.6 msec typical
Track-to-track seek, write	0.9 msec typical	0.9 msec typical
Full disc seek, read	12 msec typical	12 msec typical
Full disc seek, write	13 msec typical	13 msec typical
Average Latency	2.99 msec	2.99 msec
Other		
Default buffer (cache) size	1,024 Kbytes	1,024 Kbytes
Spindle Speed	10,000 RPM	10,016 RPM
Nonrecoverable error rate	1 per 10^{15} bits read	1 per 10^{15} bits read
Seek errors (SCSI)	1 per 10^{8} bits read	1 per 10^{8} bits read

Courtesy Seagate Technology © 1999.

soon after the data transfer is completed. Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called **fixed-head** disks, whereas disk units with an actuator are called **movable-head**

Petitioner Uber Ex-1052, 0142

disks. For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is quite high, so fixed-head disks are not commonly used.

A **disk controller,** typically embedded in the disk drive, controls the disk drive and interfaces it to the computer system. One of the standard interfaces used today for disk drives on PC and workstations is called SCSI (Small Computer Storage Interface). The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to take place. To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the **seek time.** Typical seek times are 12 to 14 msec on desktops and 8 or 9 msecs on servers. Following that, there is another delay—called the **rotational delay** or **latency**—while the beginning of the desired block rotates into position under the read/write head. Finally, some additional time is needed to transfer the data; this is called the **block transfer time.** Hence, the total time needed to locate and transfer an arbitrary block, given its address, is the sum of the seek time, rotational delay, and block transfer time. The seek time and rotational delay are usually much larger than the block transfer time. To make the transfer of multiple blocks more efficient, it is common to transfer several consecutive blocks on the same track or cylinder. This eliminates the seek time and rotational delay for all but the first block and can result in a substantial saving of time when numerous contiguous blocks are transferred. Usually, the disk manufacturer provides a **bulk transfer rate** for calculating the time required to transfer consecutive blocks. Appendix B contains a discussion of these and other disk parameters.

The time needed to locate and transfer a disk block is in the order of milliseconds, usually ranging from 12 to 60 msec. For contiguous blocks, locating the first block takes from 12 to 60 msec, but transferring subsequent blocks may take only 1 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on disk. In any case, a transfer time in the order of milliseconds is considered quite high compared with the time required to process data in main memory by current CPUs. Hence, locating data on disk is a *major bottleneck* in database applications. The file structures we discuss here and in Chapter 6 attempt to *minimize the number of block transfers* needed to locate and transfer the required data from disk to main memory.

5.2.2 Magnetic Tape Storage Devices

Disks are **random access** secondary storage devices, because an arbitrary disk block may be accessed "at random" once we specify its address. Magnetic tapes are **sequential access** devices; to access the nth block on tape, we must first scan over the preceding $n - 1$ blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audio or video tapes. A **tape drive** is required to read the data from or to write the data to a **tape reel.** Usually, each group of bits that forms a byte is stored across the tape, and the bytes themselves are stored consecutively on the tape.

A read/write head is used to read or write data on tape. Data records on tape are also stored in blocks—although the blocks may be substantially larger than those for disks, and interblock gaps are also quite large. With typical tape densities of 1600 to 6250 bytes

Petitioner Uber Ex-1052, 0143

per inch, a typical interblock gap[5] of 0.6 inches corresponds to 960 to 3750 bytes of wasted storage space. For better space utilization it is customary to group many records together in one block.

The main characteristic of a tape is its requirement that we access the data blocks in **sequential order.** To get to a block in the middle of a reel of tape, the tape is mounted and then scanned until the required block gets under the read/write head. For this reason, tape access can be slow and tapes are not used to store on-line data, except for some specialized applications. However, tapes serve a very important function—that of **backing up** the database. One reason for backup is to keep copies of disk files in case the data is lost because of a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape. Tapes can also be used to store excessively large database files. Finally, database files that are seldom used or outdated but are required for historical record keeping can be **archived** on tape. Recently, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 Gbytes, as well as 4-mm helical scan data cartridges and CD-ROMs (compact disks–read only memory) have become popular media for backing up data files from workstations and personal computers. They are also used for storing images and system libraries. In the next section we review the recent development in disk storage technology called RAID.

5.3 Parallelizing Disk Access Using RAID Technology

With the exponential growth in the performance and capacity of semiconductor devices and memories, faster microprocessors with larger and larger primary memories are continually becoming available. To match this growth, it is natural to expect that secondary storage technology must also take steps to keep up in performance and reliability with processor technology.

A major advance in secondary storage technology is represented by the development of RAID, which originally stood for **Redundant Arrays of Inexpensive Disks.** Lately, the "I" in RAID is said to stand for Independent. The RAID idea received a very positive endorsement by industry and has been developed into an elaborate set of alternative RAID architectures (RAID levels 0 through 6). We highlight the main features of the technology below.

The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.[6] While RAM capacities have quadrupled every two to three years, disk *access times* are improving at less than

5. Called *interrecord gaps* in tape terminology.

6. This was predicted by Gordon Bell to be about 40 percent every year between 1974 and 1984 and is now supposed to exceed 50 percent per year.

Petitioner Uber Ex-1052, 0144

Table 5.2 Trends in Disk Technology

	1993 Parameter Values*	Historical Rate of Improvement per Year (%)*	Expected 1999 Values**
Areal density	50–150 Mbits/sq. inch	27	2–3 GB/sq. inch
Linear density	40,000–60,000 bits/inch	13	238 Kbits/inch
Inter-track density	1,500–3,000 tracks/inch	10	11550 tracks/inch
Capacity (3.5" form factor)	100–2000 MB	27	36 GB
Transfer rate	3–4 MB/s	22	17–28 MB/sec
Seek time	7–20 ms	8	5–7 msec

**Source:* From Chen, Lee, Gibson, Katz and Patterson (1994), *ACM Computing Surveys*, Vol. 26, No. 2 (June 1994). Reprinted by permission.

***Source:* IBM Ultrastar 36XP and 18ZX hard disk drives.

10 percent per year, and disk *transfer rates* are improving at roughly 20 percent per year. Disk *capacities* are indeed improving at more than 50 percent per year, but the speed and access time improvements are of a much smaller magnitude. Table 5.2 shows trends in disk technology in terms of 1993 parameter values and rates of improvement.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving processing of video, audio, image, and spatial data (see Chapters 23 and 27 for details of these applications), with corresponding lack of fast access to large, shared data sets.

The natural solution is a large array of small independent disks acting as a single higher-performance logical disk. A concept called **data striping** is used, which utilizes *parallelism* to improve disk performance. Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk. Figure 5.3 shows a file distributed or *striped* over four disks. Striping improves overall I/O performance by allowing multiple I/Os to be serviced in parallel, thus providing high overall transfer rates. Data striping also accomplishes load balancing among disks. Moreover, by storing redundant information on disks using parity or some other error correction code, reliability can be improved. In Sections 5.3.1 and 5.3.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 5.3.3 discusses RAID organizations.

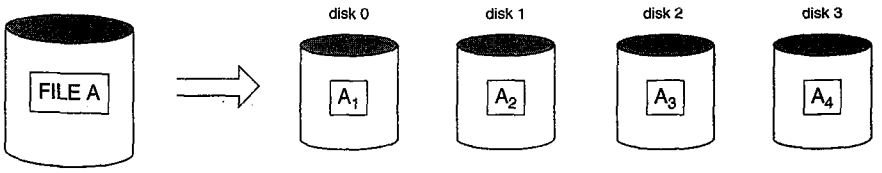


Figure 5.3 Data striping. File A is striped across four disks.

Petitioner Uber Ex-1052, 0145

5.3.1 Improving Reliability with RAID

For an array of n disks, the likelihood of failure is n times as much as that for one disk. Hence, if the MTTF (Mean Time To Failure) of a disk drive is assumed to be 200,000 hours or about 22.8 years (typical times range up to 1 million hours), that of a bank of 100 disk drives becomes only 2000 hours or 83.3 days. Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

One technique for introducing redundancy is called **mirroring** or **shadowing.** Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours, then the mean time to data loss of a mirrored disk system using 100 disks with MTTF of 200,000 hours each is $(200{,}000)^2/(2 * 24) = 8.33 * 10^8$ hours, which is 95,028 years.[7] Disk mirroring also doubles the rate at which read requests are handled, since a read can go to either disk. The transfer rate of each read, however, remains the same as that for a single disk.

Another solution to the problem of reliability is to store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure. The incorporation of redundancy must consider two problems: (1) selecting a technique for computing the redundant information, and (2) selecting a method of distributing the redundant information across the disk array. The first problem is addressed by using error correcting codes involving parity bits, or specialized codes such as Hamming codes. Under the parity scheme, a redundant disk may be considered as having the sum of all the data in the other disks. When a disk fails, the missing information can be constructed by a process similar to subtraction.

For the second problem, the two major approaches are either to store the redundant information on a small number of disks or to distribute it uniformly across all disks. The latter results in better load balancing. The different levels of RAID choose a combination of these options to implement redundancy, and hence to improve reliability.

5.3.2 Improving Performance with RAID

The disk arrays employ the technique of data striping to achieve higher transfer rates. Note that data can be read or written only one block at a time, so a typical transfer contains 512 bytes. Disk striping may be applied at a finer granularity by breaking up a byte of data into bits and spreading the bits to different disks. Thus, **bit-level data striping** consists of splitting a byte of data and writing bit j to the jth disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate. Each disk participates in each I/O request and the total amount of data read

7. The formulas for MTTF calculations appear in Chen et al. (1994).

Petitioner Uber Ex-1052, 0146

per request is eight times as much. Bit-level striping can be generalized to a number of disks that is either a multiple or a factor of eight. Thus, in a four-disk array, bit n goes to the disk which is (n mod 4).

The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to **block-level striping.** Figure 5.3 shows block-level data striping assuming the data file contained four blocks. With block-level striping, multiple independent requests that access single blocks (small requests) can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests. Requests that access multiple blocks (large requests) can be parallelized, thus reducing their response time. In general, the more the number of disks in an array, the larger the potential performance benefit. However, assuming independent failures, the disk array of 100 disks collectively has a $1/100^{th}$ the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

5.3.3 RAID Organizations and Levels

Different RAID organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. In the initial proposal, levels 1 through 5 of RAID were proposed, and two additional levels—0 and 6—were added later.

RAID level 0 has no redundant data and hence has the best write performance since updates do not have to be duplicated. However, its read performance is not as good as RAID level 1, which uses mirrored disks. In the latter, performance improvement is possible by scheduling a read request to the disk with shortest expected seek and rotational delay. RAID level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Thus, in one particular version of this level, three redundant disks suffice for four original disks whereas, with mirroring—as in level 1—four would be required. Level 2 includes both error detection and correction, although detection is generally not required because broken disks identify themselves.

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed. Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Finally, RAID level 6 applies the so-called $P + Q$ redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks. The seven RAID levels (0 through 6) are illustrated in Figure 5.4 schematically.

Rebuilding in case of disk failure is easiest for RAID level 1. Other levels require the reconstruction of a failed disk by reading multiple disks. Level 1 is used for critical applications such as storing logs of transactions. Levels 3 and 5 are preferred for large volume storage, with level 3 providing higher transfer rates. Designers of a RAID setup for a given application mix have to confront many design decisions such as the level of RAID, the number of disks, the choice of parity schemes, and grouping of disks for block-level striping. Detailed performance studies on small reads and writes (referring to I/O requests for one striping unit) and large reads and writes (referring to I/O requests for one stripe unit from each disk in an error-correction group) have been performed.

Petitioner Uber Ex-1052, 0147

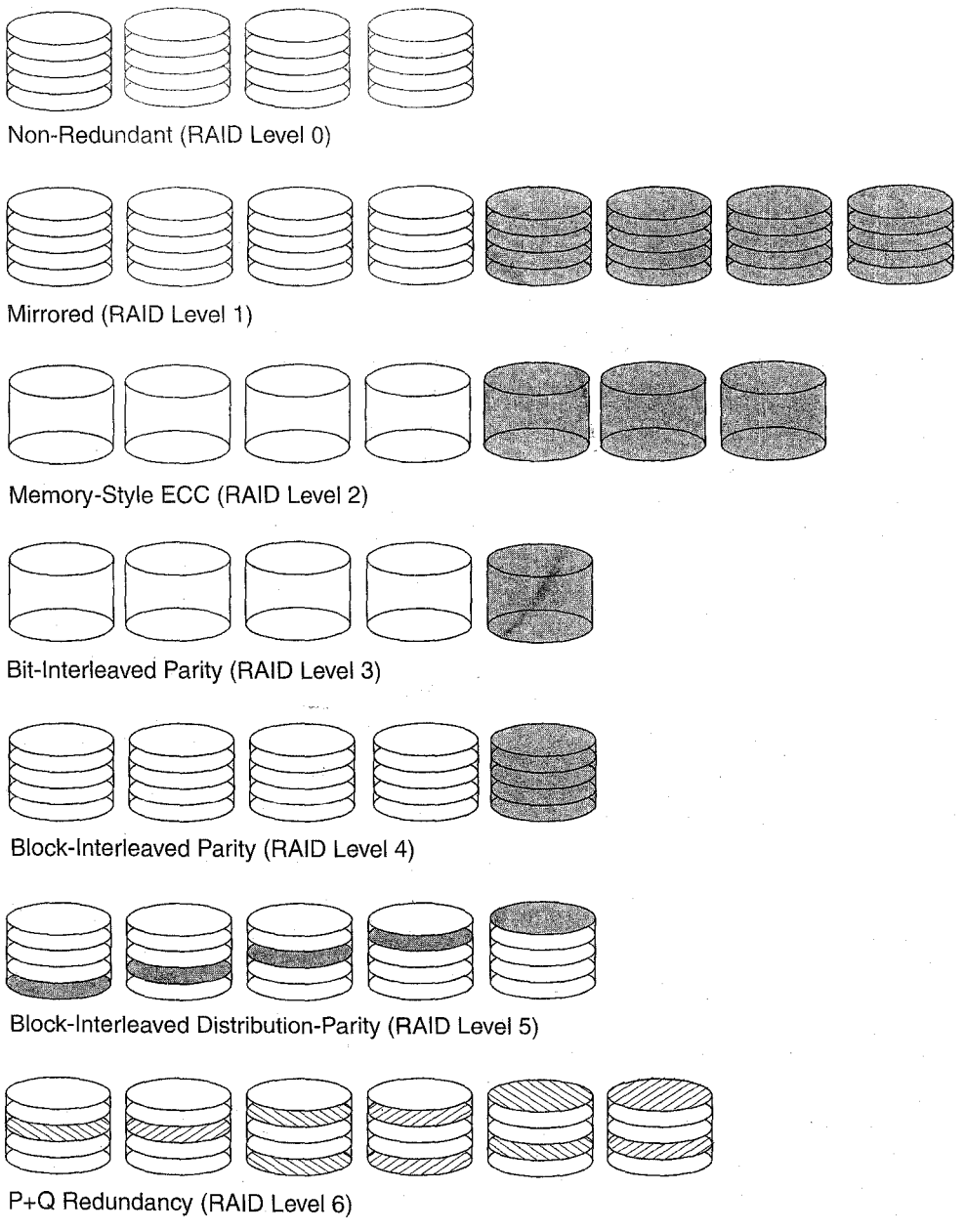


Figure 5.4 Multiple levels of RAID. From Chen, Lee, Gibson, Katz, and Patterson (1994), ACM Computing Survey, Vol. 26, No. 2 (June 1994). Reprinted with permisson.

Petitioner Uber Ex-1052, 0148

5.4 Buffering of Blocks

When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer. This is possible because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

Figure 5.5 illustrates how two processes can proceed in parallel. Processes A and B are running **concurrently** in an **interleaved** fashion, whereas processes C and D are running **concurrently** in a **parallel** fashion. When a single CPU controls multiple processes, parallel execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when processes can run concurrently in a parallel fashion, either because a separate disk I/O processor is available or because multiple CPU processors exist.

Figure 5.6 illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer. The CPU can start processing a block once its transfer to main memory is completed; at the same time the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called **double buffering** and can also be used to write a continuous stream of blocks from memory to the disk. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which

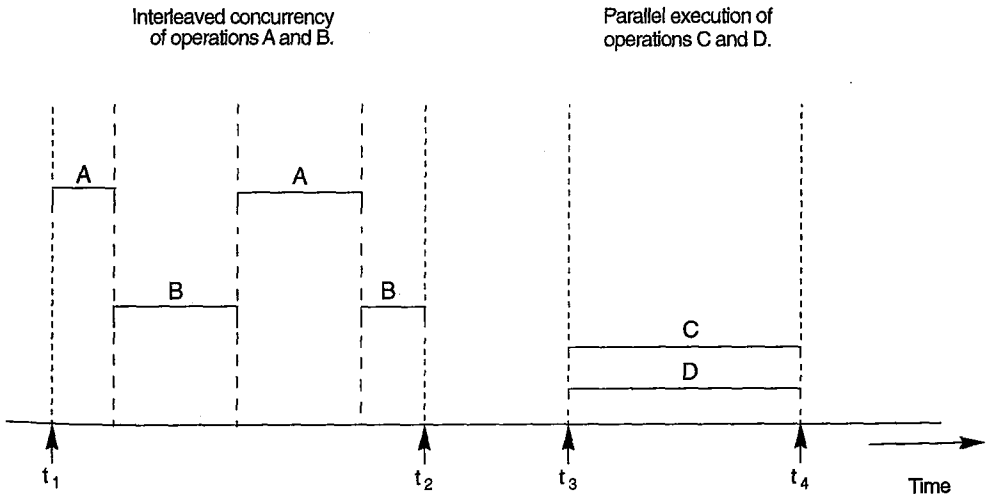


Figure 5.5 Interleaved concurrency versus parallel execution.

Petitioner Uber Ex-1052, 0149

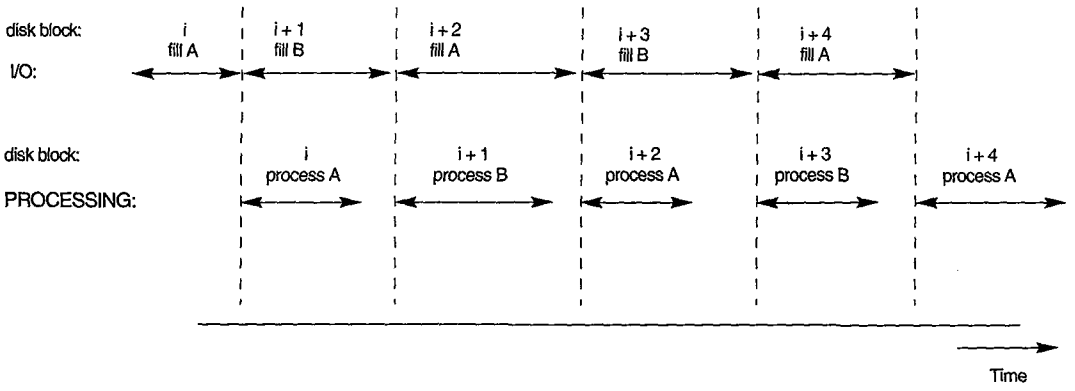


Figure 5.6 Use of two buffers, A and B, for reading from disk.

eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

5.5 Placing File Records on Disk

In this section we define the concepts of records, record types, and files. We then discuss techniques for placing file records on disk.

5.5.1 Records and Record Types

Data is usually stored in the form of **records.** Each record consists of a collection of related data **values** or **items,** where each value is formed of one or more bytes and corresponds to a particular **field** of the record. Records usually describe entities and their attributes. For example, an EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as NAME, BIRTHDATE, SALARY, or SUPERVISOR. A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition. A **data type,** associated with each field, specifies the type of values a field can take.

The data type of a field is usually one of the standard data types used in programming. These include numeric (integer, long integer, or floating point), string of characters (fixed-length or varying), Boolean (having 0 and 1 or TRUE and FALSE values only), and sometimes specially coded **date** and **time** data types. The number of bytes required for each data type is fixed for a given computer system. An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 10 bytes (assuming a format of YYYY-MM-DD), and a fixed-length string of k characters k bytes. Variable-length strings may require as many bytes as there are characters in each field value. For example,

Petitioner Uber Ex-1052, 0150

an EMPLOYEE record type may be defined—using the C programming language notation—as the following structure:

```
struct employee{
   char name[30];
   char ssn[9];
   int salary;
   int jobcode;
   char department[20];
};
```

In recent database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as BLOBs (Binary Large Objects). A BLOB data item is typically stored separately from its record in a pool of disk blocks, and a pointer to the BLOB is included in the record.

5.5.2 Files, Fixed-Length Records, and Variable-Length Records

A **file** is a *sequence* of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records.** If different records in the file have different sizes, the file is said to be made up of **variable-length records.** A file may have variable-length records for several reasons:

- The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the NAME field of EMPLOYEE can be a variable-length field.
- The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a **repeating group.**
- The file records are of the same record type, but one or more of the fields are **optional;** that is, they may have values for some but not all of the file records (**optional fields**).
- The file contains records of *different record types* and hence of varying size (**mixed file**). This would occur if related records of different types were *clustered* (placed together) on disk blocks; for example, the GRADE_REPORT records of a particular student may be placed following that STUDENT's record.

The fixed-length EMPLOYEE records in Figure 5.7(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files. Notice that it is possible to represent a file that logically should have variable-length records as a fixed-length records file. For example, in the case of optional fields we could have *every field* included in *every*

Petitioner Uber Ex-1052, 0151

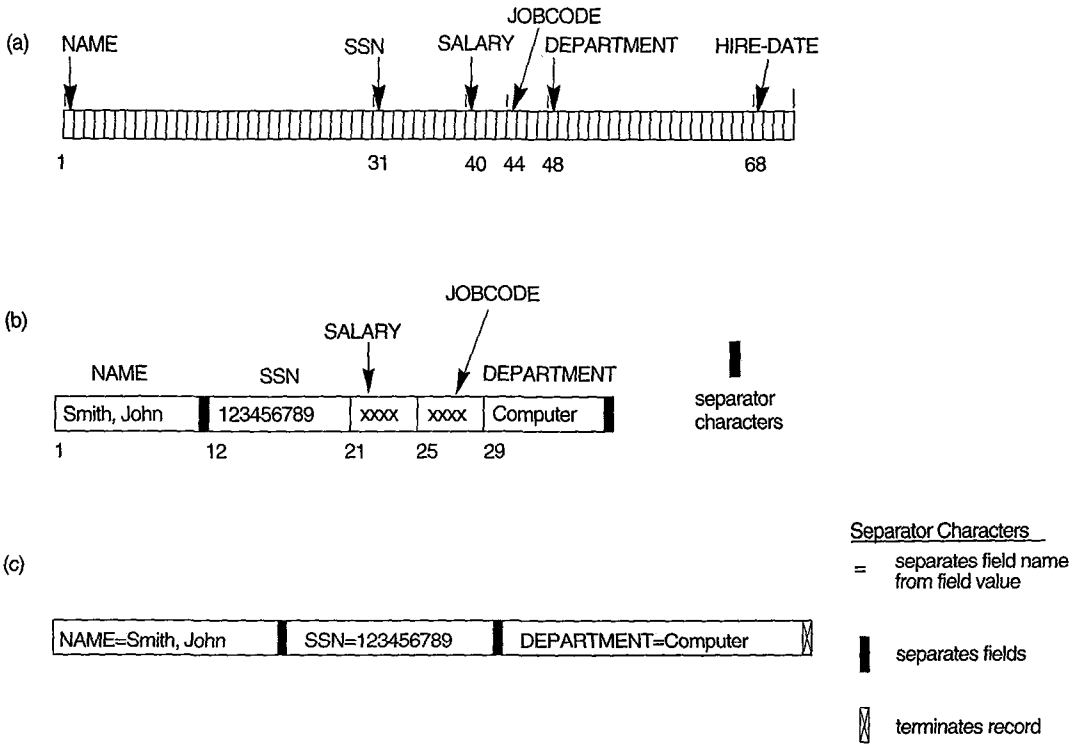


Figure 5.7 Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

file record but store a special null value if no value exists for that field. For a repeating field, we could allocate as many spaces in each record as the *maximum number of values* that the field can take. In either case, space is wasted when certain records do not have values for all the physical spaces provided in each record. We now consider other options for formatting records of a file of variable-length records.

For *variable-length fields*, each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special **separator** characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields (Figure 5.7b), or we can store the length in bytes of the field in the record, preceding the field value.

A file of records with *optional fields* can be formatted in different ways. If the total number of fields for the record type is large but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values. Three types of separator characters are used in Figure 5.7(c), although we could use the same separator character for the first two purposes—separating the field name from the field value and separating one field from the

Petitioner Uber Ex-1052, 0152

next field. A more practical option is to assign a short **field type** code—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs.

A *repeating field* needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field. Finally, for a file that includes *records of different types*, each record is preceded by a **record type** indicator. Understandably, programs that process files of variable-length records—which are usually part of the file system and hence hidden from the typical programmers—need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.[8]

5.5.3 Record Blocking and Spanned Versus Unspanned Records

The records of a file must be allocated to disk blocks because a block is the *unit of data transfer* between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block. Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit bfr = $\lfloor B/R \rfloor$ records per block, where the $\lfloor (x) \rfloor$ (*floor function*) *rounds down* the number x to an integer. The value *bfr* is called the **blocking factor** for the file. In general, R may not divide B exactly, so we have some unused space in each block equal to

$B - (bfr * R)$ bytes

To utilize this unused space, we can store part of a record on one block and the rest on another. A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called **spanned,** because records can span more than one block. Whenever a record is larger than a block, we *must* use a spanned organization. If records are not allowed to cross block boundaries, the organization is called **unspanned.** This is used with fixed-length records having $B > R$ because it makes each record start at a known location in the block, simplifying record processing. For variable-length records, either a spanned or an unspanned organization can be used. If the average record is large, it is advantageous to use spanning to reduce the lost space in each block. Figure 5.8 illustrates spanned versus unspanned organization.

For variable-length records using spanned organization, each block may store a different number of records. In this case, the blocking factor *bfr* represents the *average* number of records per block for the file. We can use *bfr* to calculate the number of blocks b needed for a file of r records:

$b = \lceil (r/bfr) \rceil$ blocks

where the $\lceil (x) \rceil$ (*ceiling function*) rounds the value x up to the next integer.

8. Other schemes are also possible for representing variable-length records.

Petitioner Uber Ex-1052, 0153

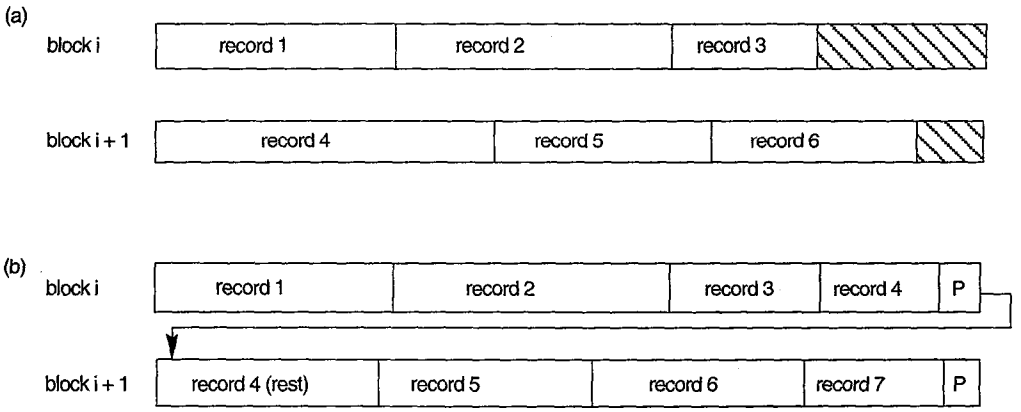


Figure 5.8 Types of record organization. (a) Unspanned. (b) Spanned.

5.5.4 Allocating File Blocks on Disk

There are several standard techniques for allocating the blocks of a file on disk. In **contiguous allocation** the file blocks are allocated to consecutive disk blocks. This makes reading the whole file very fast using double buffering, but it makes expanding the file difficult. In **linked allocation** each file block contains a pointer to the next file block. This makes it easy to expand the file but makes it slow to read the whole file. A combination of the two allocates **clusters** of consecutive disk blocks, and the clusters are linked. Clusters are sometimes called **file segments** or **extents.** Another possibility is to use **indexed allocation,** where one or more **index blocks** contain pointers to the actual file blocks. It is also common to use combinations of these techniques.

5.5.5 File Headers

A **file header** or **file descriptor** contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, using the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a **linear search** through the file blocks. Each file block is copied into a buffer and searched either until the record is located or all the file blocks have been searched unsuccessfully. This can be very time-consuming for a large file. The goal of a good file organization is to locate the block that contains a desired record with a minimal number of block transfers.

Petitioner Uber Ex-1052, 0154

5.6 Operations on Files

Operations on files are usually grouped into **retrieval operations** and **update operations.** The former do not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to **select** one or more records for retrieval, deletion, or modification based on a **selection condition** (or **filtering condition**), which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields NAME, SSN, SALARY, JOBCODE, and DEPARTMENT. A **simple selection condition** may involve an equality comparison on some field value—for example, (SSN = '123456789') or (DEPARTMENT = 'Research'). More complex conditions can involve other types of comparison operators, such as > or ≥; an example is (SALARY ≥ 30000). The general case is to have an arbitrary Boolean expression on the fields of the file as the selection condition.

Search operations on files are generally based on simple selection conditions. A complex condition must be decomposed by the DBMS (or the programmer) to extract a simple condition that can be used to locate the records on disk. Each located record is then checked to determine whether it satisfies the full selection condition. For example, we may extract the simple condition (DEPARTMENT = 'Research') from the complex condition ((SALARY ≥ 30000) AND (DEPARTMENT = 'Research')); each record satisfying (DEPARTMENT = 'Research') is located and then tested to see if it also satisfies (SALARY ≥ 30000).

When several file records satisfy a search condition, the *first* record—with respect to the physical sequence of file records—is initially located and designated the **current record.** Subsequent search operations commence from this record and locate the *next* record in the file that satisfies the condition.

Actual operations for locating and accessing file records vary from system to system. Below, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access the records by using these commands, so we sometimes refer to **program variables** in the following descriptions:

- *Open:* Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.
- *Reset:* Sets the file pointer of an open file to the beginning of the file.
- *Find* (or *Locate*): Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the *current record*. Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.
- *Read* (or *Get*): Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.

Petitioner Uber Ex-1052, 0155

- *FindNext:* Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record.
- *Delete:* Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
- *Modify:* Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.
- *Insert:* Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.
- *Close:* Completes the file access by releasing the buffers and performing any other needed cleanup operations.

The preceding (except for Open and Close) are called **record-at-a-time** operations, because each operation applies to a single record. It is possible to streamline the operations Find, FindNext, and Read into a single operation, Scan, whose description is as follows:

- *Scan:* If the file has just been opened or reset, *Scan* returns the first record; otherwise it returns the next record. If a condition is specified with the operation, the returned record is the first or next record satisfying the condition.

In database systems, additional **set-at-a-time** higher-level operations may be applied to a file. Examples of these are as follows:

- *FindAll:* Locates *all* the records in the file that satisfy a search condition.
- *FindOrdered:* Retrieves all the records in the file in some specified order.
- *Reorganize:* Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

At this point, it is worthwhile to note the difference between the terms *file organization* and *access method*. A **file organization** refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An **access method,** on the other hand, provides a group of operations—such as those listed earlier—that can be applied to a file. In general, it is possible to apply several access methods to a file organization. Some access methods, though, can be applied only to files organized in certain ways. For example, we cannot apply an indexed access method to a file without an index (see Chapter 6).

Usually, we expect to use some search conditions more than others. Some files may be **static,** meaning that update operations are rarely performed; other, more **dynamic** files may change frequently, so update operations are constantly applied to them. A successful file organization should perform as efficiently as possible the operations we expect to *apply frequently* to the file. For example, consider the EMPLOYEE file (Figure 5.7a), which stores the records for current employees in a company. We expect to insert records (when employees

Petitioner Uber Ex-1052, 0156

are hired), delete records (when employees leave the company), and modify records (say, when an employee's salary or job is changed). Deleting or modifying a record requires a selection condition to identify a particular record or set of records. Retrieving one or more records also requires a selection condition.

If users expect mainly to apply a search condition based on SSN, the designer must choose a file organization that facilitates locating a record given its SSN value. This may involve physically ordering the records by SSN value or defining an index on SSN (see Chapter 6). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks be grouped by department. For this application, it is best to store all employee records having the same department value contiguously, clustering them into blocks and perhaps ordering them by name within each department. However, this arrangement conflicts with ordering the records by SSN values. If both applications are important, the designer should choose an organization that allows both operations to be done efficiently. Unfortunately, in many cases there may not be an organization that allows all needed operations on a file to be implemented efficiently. In such cases a compromise must be chosen that takes into account the expected importance and mix of retrieval and update operations.

In the following sections and in Chapter 6, we discuss methods for organizing records of a file on disk. Several general techniques, such as ordering, hashing, and indexing, are used to create access methods. In addition, various general techniques for handling insertions and deletions work with many file organizations.

5.7 Files of Unordered Records (Heap Files)

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a **heap** or **pile file.**[9] This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 6. It is also used to collect and store data records for future use.

Inserting a new record is *very efficient:* the last disk block of the file is copied into a buffer; the new record is added; and the block is then **rewritten** back to disk. The address of the last file block is kept in the file header. However, searching for a record using any search condition involves a **linear search** through the file block by block—an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of b blocks, this requires searching $(b/2)$ blocks, on average. If no records or several records satisfy the search condition, the program must read and search all b blocks in the file.

To delete a record, a program must first find its block, copy the block into a buffer, then delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an

9. Sometimes this organization is called a **sequential file.**

Petitioner Uber Ex-1052, 0157

extra byte or bit, called a **deletion marker,** stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value of the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic **reorganization** of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records. After such a reorganization, the blocks are filled to capacity once more. Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations.

We can use either spanned or unspanned organization for an unordered file, and it may be used with either fixed-length or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a modified record, because the modified record may not fit in its old space on disk.

To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special techniques for **external sorting** are used (see Chapter 18).

For a file of unordered *fixed-length records* using *unspanned blocks* and *contiguous allocation,* it is straightforward to access any record by its **position** in the file. If the file records are numbered 0, 1, 2, . . . , $r - 1$ and the records in each block are numbered 0, 1, . . . , $bfr - 1$, where bfr is the blocking factor, then the ith record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the $(i \bmod bfr)^{th}$ record in that block. Such a file is often called a **relative** or **direct file** because records can easily be accessed directly by their relative positions. Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates the construction of access paths on the file, such as the indexes discussed in Chapter 6.

5.8 Files of Ordered Records (Sorted Files)

We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field.** This leads to an **ordered** or **sequential** file.[10] If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record—then the field is called the **ordering key** for the file. Figure 5.9 shows an ordered file with NAME as the ordering key field (assuming that employees have distinct names).

Ordered records have some advantages over unordered files. First, reading the records in order of the ordering key values becomes extremely efficient, because no sorting is required. Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses, because the next record is in the same block as the current one (unless the current record is the last one in the block). Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files.

10. The term *sequential file* has also been used to refer to unordered files.

Petitioner Uber Ex-1052, 0158

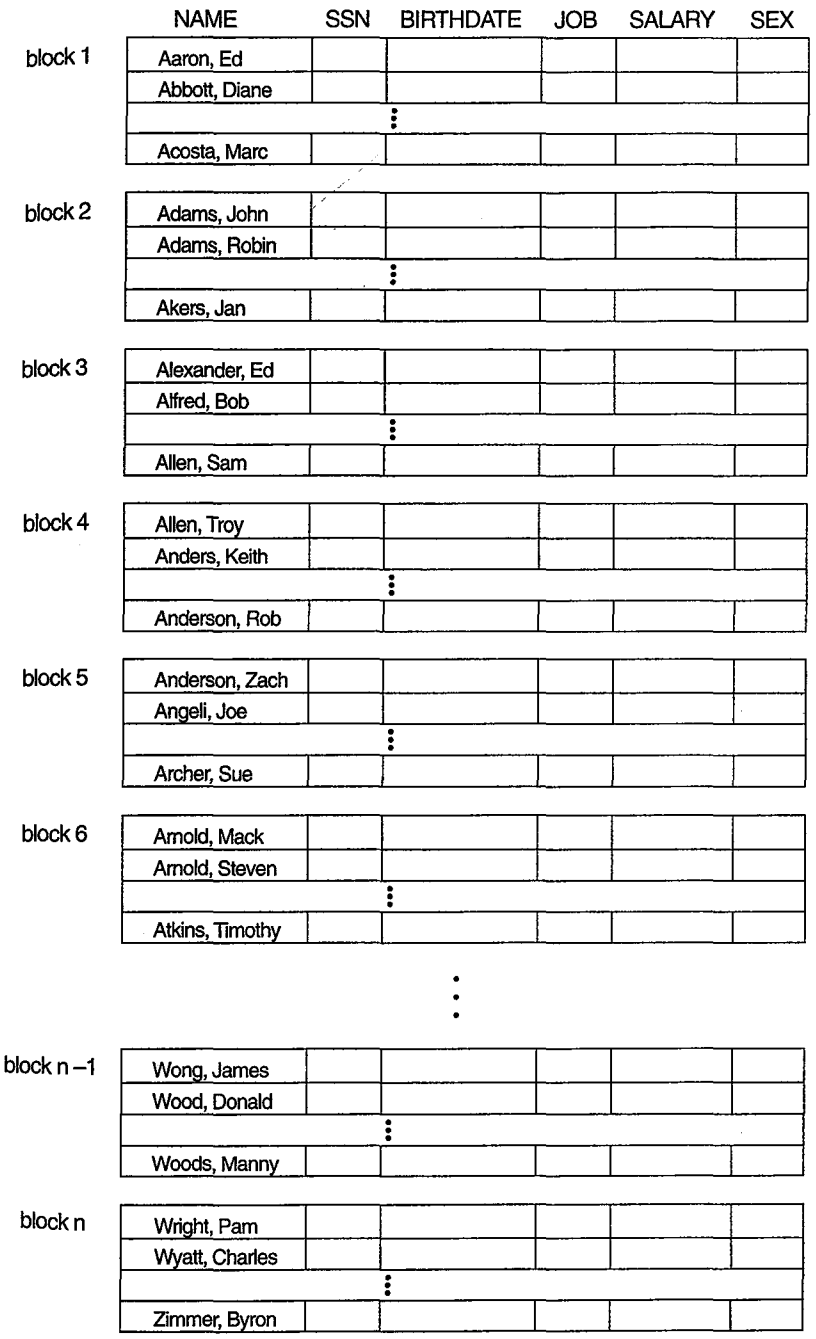


Figure 5.9 Some blocks of an ordered (sequential) file of EMPLOYEE records with NAME as the ordering key field.

Petitioner Uber Ex-1052, 0159

A **binary search** for disk files can be done on the blocks rather than on the records. Suppose that the file has b blocks numbered 1, 2, . . . , b; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K. Assuming that disk addresses of the file blocks are available in the file header, the binary search can be described by Algorithm 5.1. A binary search usually accesses $\log_2(b)$ blocks, whether the record is found or not—an improvement over linear searches, where, on the average, $(b/2)$ blocks are accessed when the record is found and b blocks are accessed when the record is not found.

ALGORITHM 5.1 Binary search on an ordering key of a disk file.

```
l ← 1; u ← b; (* b is the number of file blocks*)
while (u ≥ l) do
  begin i ← (l + u) div 2;
  read block i of the file into the buffer;
  if K < (ordering key field value of the first record in block i)
  then u ← i − 1
  else if K > (ordering key field value of the last record in block i)
      then l ← i + 1
      else if the record with ordering key field value = K is in the buffer
         then goto found
         else goto notfound;
  end;
goto notfound;
```

A search criterion involving the conditions >, <, ≥, and ≤ on the ordering field is quite efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 5.9, if the search criterion is (NAME < 'G')—where < means *alphabetically before*—the records satisfying the search criterion are those from the beginning of the file up to the first record that has a NAME value starting with the letter G.

Ordering does not provide any advantages for random or ordered access of the records based on values of the other *nonordering fields* of the file. In these cases we do a linear search for random access. To access the records in order based on a nonordering field, it is necessary to create another sorted copy—in a different order—of the file.

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time-consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

One option for making insertion more efficient is to keep some unused space in each block for new records. However, once this space is used up, the original problem resurfaces. Another frequently used method is to create a temporary *unordered* file called an

Petitioner Uber Ex-1052, 0160

overflow or **transaction** file. With this technique, the actual ordered file is called the **main** or **master** file. New records are inserted at the end of the overflow file rather than in their correct position in the main file. Periodically, the overflow file is sorted and merged with the master file during file reorganization. Insertion becomes very efficient, but at the cost of increased complexity in the search algorithm. The overflow file must be searched using a linear search if, after the binary search, the record is not found in the main file. For applications that do not require the most up-to-date information, overflow records can be ignored during a search.

Modifying a field value of a record depends on two factors: (1) the search condition to locate the record and (2) the field to be modified. If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search. A nonordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file, which requires deletion of the old record followed by insertion of the modified record.

Reading the file records in order of the ordering field is quite efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in this case, we can first reorganize the file, and then read its blocks sequentially. To reorganize the file, first sort the records in the overflow file, and then merge them with the master file. The records marked for deletion are removed during the reorganization.

Ordered files are rarely used in database applications unless an additional access path, called a **primary index,** is used; this results in an **indexed-sequential file.** This further improves the random access time on the ordering key field. We discuss indexes in Chapter 6.

5.9 Hashing Techniques

Another type of primary file organization is based on hashing, which provides very fast access to records on certain search conditions. This organization is usually called a **hash file.**[11] The search condition must be an equality condition on a single field, called the **hash field** of the file. In most cases, the hash field is also a key field of the file, in which case it is called the **hash key.** The idea behind hashing is to provide a function h, called a **hash function** or **randomizing function,** that is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field. We describe the use of hashing for internal files in Section 5.9.1; then we show how it is modified to store external files on disk in Section 5.9.2. In Section 5.9.3 we discuss techniques for extending hashing to dynamically growing files.

11. A hash file has also been called a *direct file*.

Petitioner Uber Ex-1052, 0161

5.9.1 Internal Hashing

For internal files, hashing is typically implemented as a **hash table** through the use of an array of records. Suppose that the array index range is from 0 to M − 1 (Figure 5.10a); then we have M **slots** whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and M − 1. One common hash function is the $h(K) = K \bmod M$ function, which returns the remainder of

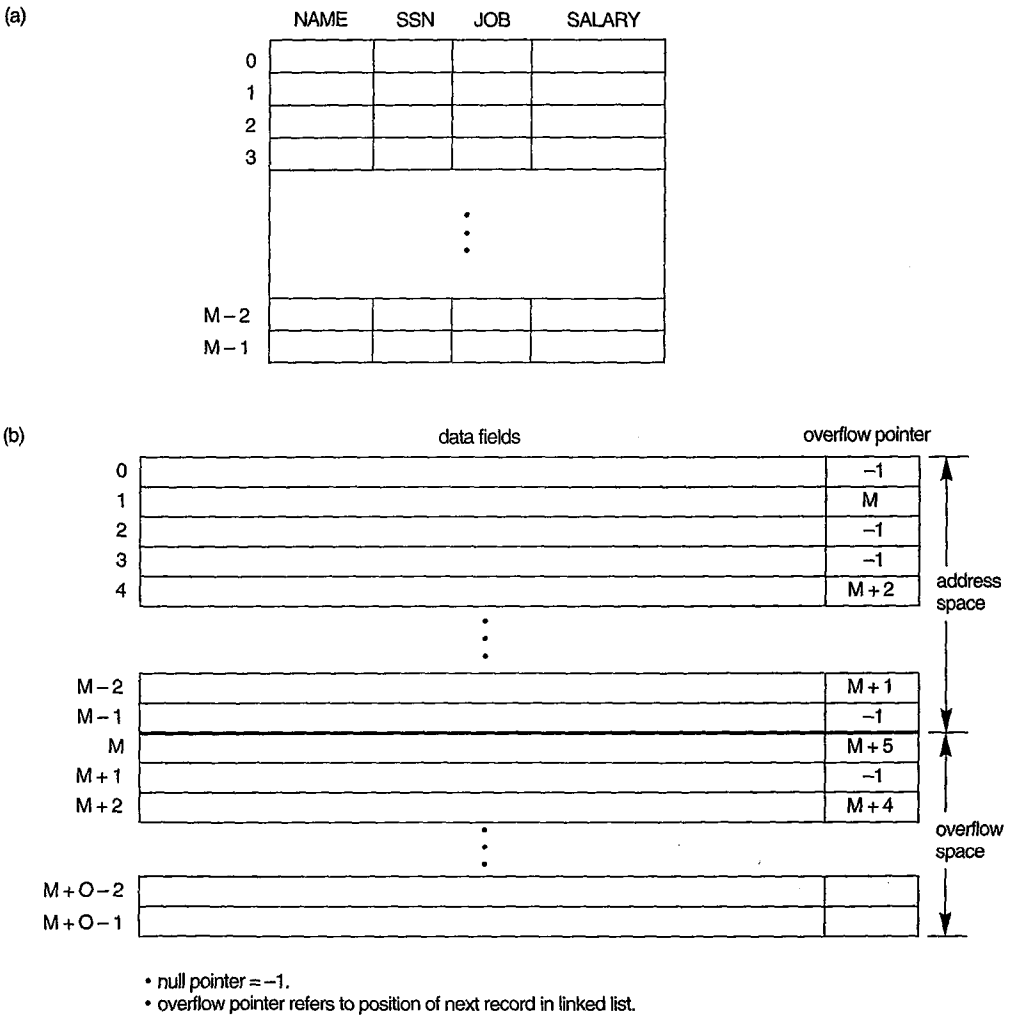


Figure 5.10 Internal hashing data structures. (a) Array of M positions for use in internal hashing. (b) Collision resolution by chaining records.

Petitioner Uber Ex-1052, 0162

an integer hash field value K after division by M; this value is then used for the record address.

Noninteger hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values. For a hash field whose data type is a string of 20 characters, Algorithm 5.2(a) can be used to calculate the hash address. We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type K: *array [1..20] of char* (in PASCAL) or *char K[20]* (in C).

ALGORITHM 5.2 Two simple hashing algorithms. (a) Applying the mod hash function to a character string K. (b) Collision resolution by open addressing.

```
(a) temp ← 1;
    for i ← 1 to 20 do temp ← temp * code(K[i]) mod M;
    hash_address ← temp mod M;
(b) i ← hash_address(K); a ← i;
    if location i is occupied
        then begin i ← (i + 1) mod M;
           while (i ≠ a) and location i is occupied
               do i ← (i + 1) mod M;
           if (i = a) then all positions are full
               else new_hash_address ← i;
           end;
```

Other hashing functions can be used. One technique, called **folding,** involves applying an arithmetic function such as *addition* or a logical function such as *exclusive or* to different portions of the hash field value to calculate the hash address. Another technique involves picking some digits of the hash field value—for example, the third, fifth, and eighth digits—to form the hash address.[12] The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because the **hash field space**—the number of possible values a hash field can take—is usually much larger than the **address space**—the number of available addresses for records. The hashing function maps the hash field space to the address space.

A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called **collision resolution.** There are numerous methods for collision resolution, including the following:

- *Open addressing:* Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found. Algorithm 5.2(b) may be used for this purpose.

12. A detailed discussion of hashing functions is outside the scope of our presentation.

Petitioner Uber Ex-1052, 0163

- *Chaining:* For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained, as shown in Figure 5.10(b).
- *Multiple hashing:* The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The algorithms for chaining are the simplest. Deletion algorithms for open addressing are rather tricky. Data structures textbooks discuss internal hashing algorithms in more detail.

The goal of a good hashing function is to distribute the records uniformly over the address space so as to minimize collisions while not leaving many unused locations. Simulation and analysis studies have shown that it is usually best to keep a hash table between 70 and 90 percent full so that the number of collisions remains low and we do not waste too much space. Hence, if we expect to have r records to store in the table, we should choose M locations for the address space such that (r/M) is between 0.7 and 0.9. It may also be useful to choose a prime number for M, since it has been demonstrated that this distributes the hash addresses better over the address space when the mod hashing function is used. Other hash functions may require M to be a power of 2.

5.9.2 External Hashing for Disk Files

Hashing for disk files is called **external hashing.** To suit the characteristics of disk storage, the target address space is made of **buckets,** each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous blocks. The hashing function maps a key into a relative bucket number, rather than assign an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in Figure 5.11.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket. We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket, as shown in Figure 5.12. The pointers in the linked list should be **record pointers,** which include both a block address and a relative record position within the block.

Hashing provides the fastest possible access for retrieving an arbitrary record given the value of its hash field. Although most good hash functions do not maintain records in order of hash field values, some functions—called **order preserving**—do. A simple example of an order preserving hash function is to take the leftmost three digits of an invoice number field as the hash address and keep the records sorted by invoice number within

Petitioner Uber Ex-1052, 0164

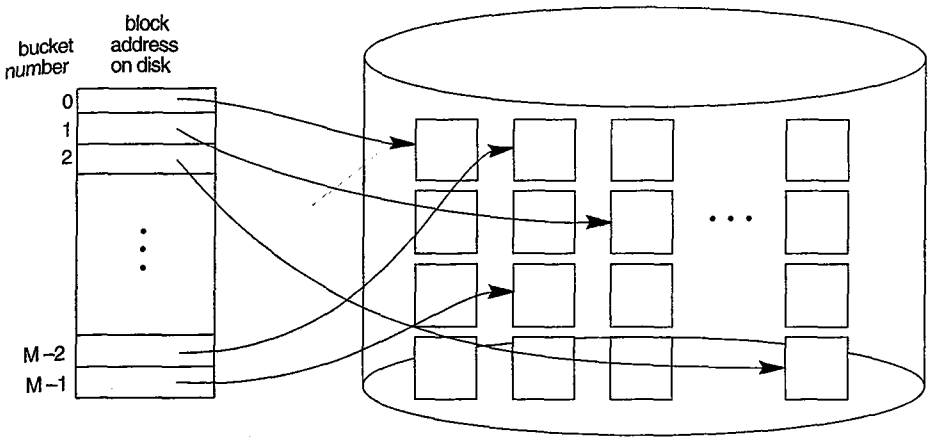


Figure 5.11 Matching bucket numbers to disk block addresses.

each bucket. Another example is to use an integer hash key directly as an index to a relative file, if the hash key values fill up a particular interval; for example, if employee numbers in a company are assigned as 1, 2, 3, . . . up to the total number of employees, we can use the identity hash function that maintains order. Unfortunately, this only works if keys are generated in order by some application.

The hashing scheme described is called **static hashing** because a fixed number of buckets M is allocated. This can be a serious drawback for dynamic files. Suppose that we allocate M buckets for the address space and let m be the maximum number of records that can fit in one bucket; then at most ($m * M$) records will fit in the allocated space. If the number of records turns out to be substantially fewer than ($m * M$), we are left with a lot of unused space. On the other hand, if the number of records increases to substantially more than ($m * M$), numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records. In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These reorganizations can be quite time consuming for large files. Newer dynamic file organizations based on hashing allow the number of buckets to vary dynamically with only localized reorganization (see Section 5.9.3).

When using external hashing, searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file. Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in overflow, we simply remove it from the linked list. Notice that removing an overflow record implies that we should keep track of empty positions in overflow. This is done easily by maintaining a linked list of unused overflow locations.

Petitioner Uber Ex-1052, 0165

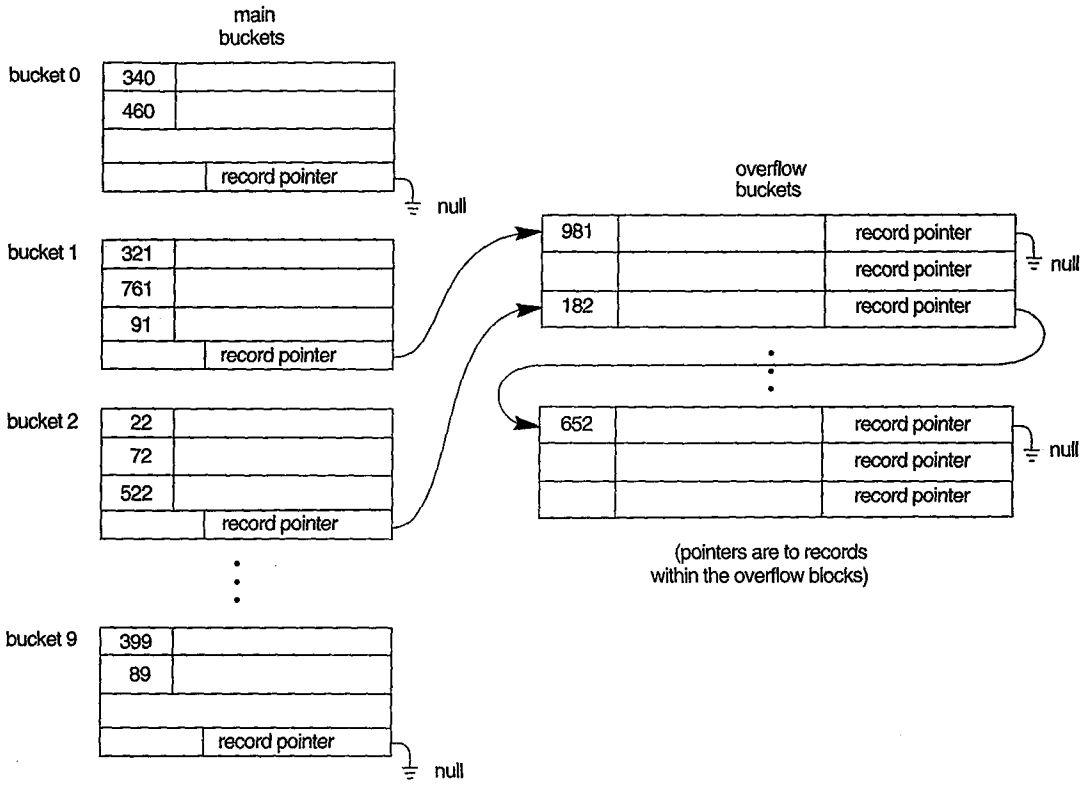


Figure 5.12 Handling overflow for buckets by chaining.

Modifying a record's field value depends on two factors: (1) the search condition to locate the record and (2) the field to be modified. If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hashing function; otherwise, we must do a linear search. A nonhash field can be modified by changing the record and rewriting it in the same bucket. Modifying the hash field means that the record can move to another bucket, which requires deletion of the old record followed by insertion of the modified record.

5.9.3 Hashing Techniques That Allow Dynamic File Expansion

A major drawback of the *static* hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The schemes described in this section attempt to remedy this situation. The first scheme—extendible hashing—stores an access structure in addition to the file, and hence is somewhat similar

Petitioner Uber Ex-1052, 0166

to indexing (Chapter 6). The main difference is that the access structure is based on the values that result after application of the hash function to the search field. In indexing, the access structure is based on the values of the search field itself. The second technique, called linear hashing, does not require additional access structures.

These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number. The access structure is built on the **binary representation** of the hashing function result, which is a string of **bits.** We call this the **hash value** of a record. Records are distributed among buckets based on the values of the *leading bits* in their hash values.

Extendible Hashing. In extendible hashing, a type of **directory**—an array of 2^d bucket addresses—is maintained, where d is called the **global depth** of the directory. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored. However, there does not have to be a distinct bucket for each of the 2^d directory locations. Several directory locations with the same first d' bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A **local depth d'**—stored with each bucket—specifies the number of bits on which the bucket contents are based. Figure 5.13 shows a directory with global depth $d = 3$.

The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket, whose local depth d' is equal to the global depth d, overflows. Halving occurs if $d > d'$ for all the buckets after some deletions occur. Most record retrievals require two block accesses—one to the directory and the other to the bucket.

To illustrate bucket splitting, suppose that a new inserted record causes overflow in the bucket whose hash values start with 01—the third bucket in Figure 5.13. The records will be distributed between two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth d' of the two new buckets is 3, which is one more than the local depth of the old bucket.

If a bucket that overflows and is split used to have a local depth d' equal to the global depth d of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 in Figure 5.13 overflows, the two new buckets need a directory with global depth $d = 4$, because the two buckets are now labeled 1110 and 1111, and hence their local depths are both 4. The directory size is hence doubled, and each of the other original locations in the directory is also split into two locations, both of which have the same pointer value as did the original location.

The main advantage of extendible hashing that makes it attractive is that the performance of the file does not degrade as the file grows, as opposed to static external hashing where collisions increase and the corresponding chaining causes additional accesses. In addition, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. The space overhead for the directory

Petitioner Uber Ex-1052, 0167

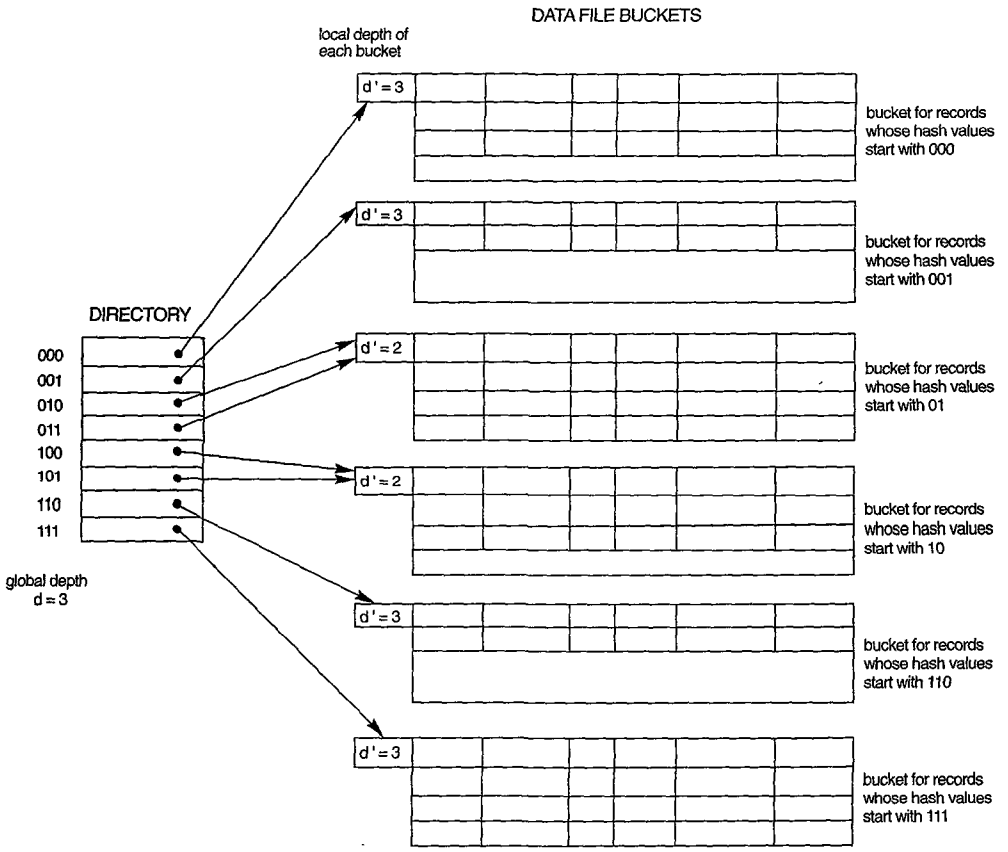


Figure 5.13 Structure of the extendible hashing scheme.

table is negligible. The maximum directory size is 2^k, where k is the number of bits in the hash value. Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time a reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and hence the scheme is considered quite desirable for dynamic files.

Linear Hashing. The idea behind linear hashing is to allow a hash file to expand and shrink its number of buckets dynamically *without* needing a directory. Suppose that the file starts with M buckets numbered 0, 1, . . . , M − 1 and uses the mod hash function $h(K) = K \bmod M$; this hash function is called the initial hash function h_i. Overflow

Petitioner Uber Ex-1052, 0168

because of collisions is still needed and can be handled by maintaining individual overflow chains for each bucket. However, when a collision leads to an overflow record in *any* file bucket, the *first* bucket in the file—bucket 0—is split into two buckets: the original bucket 0 and a new bucket M at the end of the file. The records originally in bucket 0 are distributed between the two buckets based on a different hashing function $h_{i+1}(K) = K \bmod 2M$. A key property of the two hash functions h_i and h_{i+1} is that any records that hashed to bucket 0 based on h_i will hash to either bucket 0 or bucket M based on h_{i+1}; this is necessary for linear hashing to work.

As further collisions lead to overflow records, additional buckets are split in the *linear* order 1, 2, 3, ... If enough overflows occur, all the original file buckets 0, 1, . . . , M − 1 will have been split, so the file now has 2M instead of M buckets, and all buckets use the hash function h_{i+1}. Hence, the records in overflow are eventually redistributed into regular buckets, using the function h_{i+1} via a *delayed split* of their buckets. There is no directory; only a value n—which is initially set to 0 and is incremented by 1 whenever a split occurs—is needed to determine which buckets have been split. To retrieve a record with hash key value K, first apply the function h_i to K; if $h_i(K) < n$, then apply the function h_{i+1} on K because the bucket is already split. Initially, $n = 0$, indicating that the function h_i applies to all buckets; n grows linearly as buckets are split.

When $n = M$ after being incremented, this signifies that all the original buckets have been split and the hash function h_{i+1} applies to all records in the file. At this point, n is reset to 0 (zero), and any new collisions that cause overflow lead to the use of a new hashing function $h_{i+2}(K) = K \bmod 4M$. In general, a sequence of hashing functions $h_{i+j}(K) = K \bmod (2^jM)$ is used, where $j = 0, 1, 2, \ldots$; a new hashing function h_{i+j+1} is needed whenever all the buckets 0, 1, . . . , $(2^jM) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 5.3.

Splitting can be controlled by monitoring the file load factor instead of by splitting whenever an overflow occurs. In general, the **file load factor** l can be defined as $l = r/(bfr * N)$, where r is the current number of file records, bfr is the maximum number of records that can fit in a bucket, and N is the current number of file buckets. Buckets that have been split can also be recombined if the load of the file falls below a certain threshold. Blocks are combined linearly, and N is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range. Splits can be triggered when the load exceeds a certain threshold—say, 0.9—and combinations can be triggered when the load falls below another threshold—say, 0.7.

ALGORITHM 5.3 The search procedure for linear hashing.

```
if n = 0
    then m ← h_j(K) (* m is the hash value of record with hash key K *)
    else begin
       m ← h_j(K);
       if m < n then m ← h_{j+1} (K)
       end;
search the bucket whose hash value is m (and its overflow, if any);
```

Petitioner Uber Ex-1052, 0169

5.10 Other Primary File Organizations

5.10.1 Files of Mixed Records

The file organizations we have studied so far assume that all records of a particular file are of the same record type. The records could be of EMPLOYEES, PROJECTS, STUDENTS, or DEPARTMENTS, but each file contains records of only one type. In most database applications, we encounter situations in which numerous types of entities are interrelated in various ways, as we saw in Chapter 3. Relationships among records in various files can be represented by **connecting fields.**[13] For example, a STUDENT record can have a connecting field MAJORDEPT whose value gives the name of the DEPARTMENT in which the student is majoring. This MAJORDEPT field *refers* to a DEPARTMENT entity, which should be represented by a record of its own in the DEPARTMENT file. If we want to retrieve field values from two related records, we must retrieve one of the records first. Then we can use its connecting field value to retrieve the related record in the other file. Hence, relationships are implemented by **logical field references** among the records in distinct files.

File organizations in object DBMSs, as well as legacy systems such as hierarchical and network DBMSs, often implement relationships among records as **physical relationships** realized by physical contiguity (or clustering) of related records or by physical pointers. These file organizations typically assign an **area** of the disk to hold records of more than one type so that records of different types can be **physically clustered** on disk. If a particular relationship is expected to be used very frequently, implementing the relationship physically can increase the system's efficiency at retrieving related records. For example, if the query to retrieve a DEPARTMENT record and all records for STUDENTS majoring in that department is very frequent, it would be desirable to place each DEPARTMENT record and its cluster of STUDENT records contiguously on disk in a mixed file. The concept of **physical clustering** of object types is used in object DBMSs to store related objects together in a mixed file.

To distinguish the records in a mixed file, each record has—in addition to its field values—a **record type** field, which specifies the type of record. This is typically the first field in each record and is used by the system software to determine the type of record it is about to process. Using the catalog information, the DBMS can determine the fields of that record type and their sizes, in order to interpret the data values in the record.

5.10.2 B-Trees and Other Data Structures

Other data structures can be used for primary file organizations. For example, if both the record size and the number of records in a file are small, some DBMSs offer the option of a B-tree data structure as the primary file organization. We will describe B-trees in Section 6.3.1, when we discuss the use of the B-tree data structure for indexing. In general, any data structure that can be adapted to the characteristics of disk devices can be used as a primary file organization for record placement on disk.

13. The concept of foreign keys in the relational model (Chapter 7) and references among objects in object-oriented models (Chapter 11) are examples of connecting fields.

Petitioner Uber Ex-1052, 0170

5.11 Summary

We began this chapter by discussing the characteristics of memory hierarchies and then concentrated on secondary storage devices. In particular, we focused on magnetic disks because they are used most often to store on-line database files. We reviewed the recent advances in disk technology represented by RAID (Redundant Arrays of Inexpensive [Independent] Disks).

Data on disk is stored in blocks; accessing a disk block is expensive because of the seek time, rotational delay, and block transfer time. Double buffering can be used when accessing consecutive disk blocks, to reduce the average block access time. Other disk parameters are discussed in Appendix B. We presented different ways of storing records of a file on disk. Records of a file are grouped into disk blocks and can be of fixed length or variable length, spanned or unspanned, and of the same record type or mixed-types. We discussed the file header, which describes the record formats and keeps track of the disk addresses of the file blocks. Information in the file header is used by system software accessing the file records.

We then presented a set of typical commands for accessing individual file records and discussed the concept of the current record of a file. We discussed how complex record search conditions are transformed into simple search conditions that are used to locate records in the file.

Three primary file organizations were then discussed: unordered, ordered, and hashed. Unordered files require a linear search to locate records, but record insertion is very simple. We discussed the deletion problem and the use of deletion markers.

Ordered files shorten the time required to read records in order of the ordering field. The time required to search for an arbitrary record, given the value of its ordering key field, is also reduced if a binary search is used. However, maintaining the records in order makes insertion very expensive; thus the technique of using an unordered overflow file to reduce the cost of record insertion was discussed. Overflow records are merged with the master file periodically during file reorganization.

Hashing provides very fast access to an arbitrary record of a file, given the value of its hash key. The most suitable method for external hashing is the bucket technique, with one or more contiguous blocks corresponding to each bucket. Collisions causing bucket overflow are handled by chaining. Access on any nonhash field is slow, and so is ordered access of the records on any field. We then discussed two hashing techniques for files that grow and shrink in the number of records dynamically—namely, extendible and linear hashing.

Finally, we briefly discussed other possibilities for primary file organizations, such as B-trees, and files of mixed records, which implement relationships among records of different types physically as part of the storage structure.

Review Questions

5.1. What is the difference between primary and secondary storage?

5.2. Why are disks, not tapes, used to store on-line database files?

5.3. Define the following terms: *disk*, *disk pack*, *track*, *block*, *cylinder*, *sector*, *interblock gap*, *read/write head*.

Petitioner Uber Ex-1052, 0171

5.4. Discuss the process of disk initialization.
5.5. Discuss the mechanism used to read data from or write data to the disk.
5.6. What are the components of a disk block address?
5.7. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.
5.8. Describe the mismatch between processor and disk technologies.
5.9. What are the main goals of the RAID technology? How does it achieve them?
5.10. How does disk mirroring help improve reliability? Give a quantitative example.
5.11. What are the techniques used to improve performance of disks in RAID?
5.12. What characterizes the levels in RAID organization?
5.13. How does double buffering improve block access time?
5.14. What are the reasons for having variable-length records? What types of separator characters are needed for each?
5.15. Discuss the techniques for allocating file blocks on disk.
5.16. What is the difference between a file organization and an access method?
5.17. What is the difference between static and dynamic files?
5.18. What are the typical record-at-a-time operations for accessing a file? Which of these depend on the current record of a file?
5.19. Discuss the techniques for record deletion.
5.20. Discuss the advantages and disadvantages of using (a) an unordered file, (b) an ordered file, and (c) a static hash file with buckets and chaining. Which operations can be performed efficiently on each of these organizations, and which operations are expensive?
5.21. Discuss the techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?
5.22. What are mixed files used for? What are other types of primary file organizations?

Exercises

5.23. Consider a disk with the following characteristics (these are not parameters of any particular disk unit): block size $B = 512$ bytes; interblock gap size $G = 128$ bytes; number of blocks per track = 20; number of tracks per surface = 400. A disk pack consists of 15 double-sided disks.
   a. What is the total capacity of a track, and what is its useful capacity (excluding interblock gaps)?
   b. How many cylinders are there?
   c. What are the total capacity and the useful capacity of a cylinder?
   d. What are the total capacity and the useful capacity of a disk pack?
   e. Suppose that the disk drive rotates the disk pack at a speed of 2400 rpm (revolutions per minute); what are the transfer rate (*tr*) in bytes/msec and the block transfer time (*btt*) in msec? What is the average rotational delay (*rd*) in msec? What is the bulk transfer rate? (See Appendix B.)
   f. Suppose that the average seek time is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block, given its block address?

Petitioner Uber Ex-1052, 0172

g. Calculate the average time it would take to transfer 20 random blocks, and compare this with the time it would take to transfer 20 consecutive blocks using double buffering to save seek time and rotational delay.

5.24. A file has r = 20,000 STUDENT records of *fixed length*. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), MAJORDEPTCODE (4 bytes), MINORDEPTCODE (4 bytes), CLASSCODE (4 bytes, integer), and DEGREEPROGRAM (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 5.23.

a. Calculate the record size R in bytes.
b. Calculate the blocking factor *bfr* and the number of file blocks b, assuming an unspanned organization.
c. Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously, and double buffering is used; (ii) the file blocks are not stored contiguously.
d. Assume that the file is ordered by SSN; calculate the time it takes to search for a record given its SSN value, by doing a binary search.

5.25. Suppose that only 80 percent of the STUDENT records from Exercise 5.24 have a value for PHONE, 85 percent for MAJORDEPTCODE, 15 percent for MINORDEPTCODE, and 90 percent for DEGREEPROGRAM; and suppose that we use a variable-length record file. Each record has a 1-byte *field type* for each field in the record, plus the 1-byte deletion marker and a 1-byte end-of-record marker. Suppose that we use a *spanned* record organization, where each block has a 5-byte pointer to the next block (this space is not used for record storage).

a. Calculate the average record length R in bytes.
b. Calculate the number of blocks needed for the file.

5.26. Suppose that a disk unit has the following parameters: seek time s = 20 msec; rotational delay rd = 10 msec; block transfer time btt = 1 msec; block size B = 2400 bytes; interblock gap size G = 600 bytes. An EMPLOYEE file has the following fields: SSN, 9 bytes; LASTNAME, 20 bytes; FIRSTNAME, 20 bytes; MIDDLE INIT, 1 byte; BIRTHDATE, 10 bytes; ADDRESS, 35 bytes; PHONE, 12 bytes; SUPERVISORSSN, 9 bytes; DEPARTMENT, 4 bytes; JOBCODE, 4 bytes; *deletion marker*, 1 byte. The EMPLOYEE file has r = 30,000 records, fixed-length format, and unspanned blocking. Write appropriate formulas *and* calculate the following values for the above EMPLOYEE file:

a. The record size R (including the deletion marker), the blocking factor *bfr*, and the number of disk blocks b.
b. Calculate the wasted space in each disk block because of the unspanned organization.
c. Calculate the transfer rate tr and the bulk transfer rate btr for this disk unit (see Appendix B for definitions of tr and btr).
d. Calculate the average *number of block accesses* needed to search for an arbitrary record in the file, using linear search.
e. Calculate in msec the average *time* needed to search for an arbitrary record in the file, using linear search, if the file blocks are stored on consecutive disk blocks and double buffering is used.

Petitioner Uber Ex-1052, 0173

f. Calculate in msec the average *time* needed to search for an arbitrary record in the file, using linear search, if the file blocks are *not* stored on consecutive disk blocks.

g. Assume that the records are ordered via some key field. Calculate the average *number of block accesses* and the *average time* needed to search for an arbitrary record in the file, using binary search.

5.27. A PARTS file with Part# as hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order, using the hash function $h(K) = K \bmod 8$. Calculate the average number of block accesses for a random retrieval on Part#.

5.28. Load the records of Exercise 5.27 into expandable hash files based on extendible hashing. Show the structure of the directory at each step, and the global and local depths. Use the hash function $h(K) = K \bmod 128$.

5.29. Load the records of Exercise 5.27 into an expandable hash file, using linear hashing. Start with a single disk block, using the hash function $h_0 = K \bmod 2^0$, and show how the file grows and how the hash functions change as the records are inserted. Assume that blocks are split whenever an overflow occurs, and show the value of n at each stage.

5.30. Compare the file commands listed in Section 5.6 to those available on a file access method you are familiar with.

5.31. Suppose that we have an unordered file of fixed-length records that uses an unspanned record organization. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.

5.32. Suppose that we have an ordered file of fixed-length records and an unordered overflow file to handle insertion. Both files use unspanned records. Outline algorithms for insertion, deletion, and modification of a file record and for reorganizing the file. State any assumptions you make.

5.33. Can you think of techniques other than an unordered overflow file that can be used to make insertions in an ordered file more efficient?

5.34. Suppose that we have a hash file of fixed-length records, and suppose that overflow is handled by chaining. Outline algorithms for insertion, deletion, and modification of a file record. State any assumptions you make.

5.35. Can you think of techniques other than chaining to handle bucket overflow in external hashing?

5.36. Write pseudocode for the insertion algorithms for linear hashing and for extendible hashing.

5.37. Write program code to access individual fields of records under each of the following circumstances. For each case, state the assumptions you make concerning pointers, separator characters, and so forth. Determine the type of information needed in the file header in order for your code to be general in each case.

a. Fixed-length records with unspanned blocking.

b. Fixed-length records with spanned blocking.

c. Variable-length records with variable-length fields and spanned blocking.

Petitioner Uber Ex-1052, 0174

d. Variable-length records with repeating groups and spanned blocking.
e. Variable-length records with optional fields and spanned blocking.
f. Variable-length records that allow all three cases in parts c, d, and e.

5.38. Suppose that a file initially contains r = 120,000 records of R = 200 bytes each in an unsorted (heap) file. The block size B = 2400 bytes, the average seek time s = 16 ms, the average rotational latency rd = 8.3 ms and the block transfer time btt = 0.8 ms. Assume that 1 record is deleted for every 2 records added until the total number of active records is 240,000.
a. How many block transfers are needed to reorganize the file?
b. How long does it take to find a record right before reorganization?
c. How long does it take to find a record right after reorganization?

5.39. Suppose we have a sequential (ordered) file of 100,000 records where each record is 240 bytes. Assume that B = 2400 bytes, s = 16 ms, rd = 8.3 ms, and btt = 0.8 ms. Suppose we want to make X independent random record reads from the file. We could make X random block reads or we could perform one exhaustive read of the entire file looking for those X records. The question is to decide when it would be more efficient to perform one exhaustive read of the entire file than to perform X individual random reads. That is, what is the value for X when an exhaustive read of the file is more efficient than random X reads? Develop this as a function of X.

5.40. Suppose that a static hash file initially has 600 buckets in the primary area and that records are inserted that create an overflow area of 600 buckets. If we reorganize the hash file, we can assume that the overflow is eliminated. If the cost of reorganizing the file is the cost of the bucket transfers (reading and writing all of the buckets) and the only periodic file operation is the fetch operation, then how many times would we have to perform a fetch (successfully) to make the reorganization cost-effective? That is, the reorganization cost and subsequent search cost are less than the search cost before reorganization. Support your answer. Assume s = 16 ms, rd = 8.3 ms, btt = 1 ms.

5.41. Suppose we want to create a linear hash file with a file load factor of 0.7 and a blocking factor of 20 records per bucket, which is to contain 112,000 records initially.
a. How many buckets should we allocate in the primary area?
b. What should be the number of bits used for bucket addresses?

Selected Bibliography

Wiederhold (1983) has a detailed discussion and analysis of secondary storage devices and file organizations. Optical disks are described in Berg and Roth (1989) and analyzed in Ford and Christodoulakis [1991]. Flash memory is discussed by Dippert and Levy (1993). Ruemmler and Wilkes (1994) present a survey of the magnetic-disk technology. Most textbooks on databases include discussions of the material presented here. Most data structures textbooks, including Knuth (1973), discuss static hashing in more detail; Knuth has a complete discussion of hash functions and collision resolution techniques, as well as of their performance comparison. Knuth also offers a detailed discussion of techniques for sorting external

Petitioner Uber Ex-1052, 0175

files. Textbooks on file structures include Claybrook (1983), Smith and Barnes (1987), and Salzberg (1988); they discuss additional file organizations including tree structured files, and have detailed algorithms for operations on files. Additional textbooks on file organizations include Miller (1987), and Livadas (1989). Salzberg et al. (1990) describes a distributed external sorting algorithm. File organizations with a high degree of fault tolerance are described by Bitton and Gray (1988) and by Gray et al. (1990). Disk striping is proposed in Salem and Garcia Molina (1986). The first paper on redundant arrays of inexpensive disks (RAID) is by Patterson et al. (1988). Chen and Patterson (1990) and the excellent survey of RAID by Chen et al. (1994) are additional references. Grochowski and Hoyt (1996) discuss future trends in disk drives. Various formulas for the RAID architecture appear in Chen et al. (1994).

Morris (1968) is an early paper on hashing. Extendible hashing is described in Fagin et al. (1979). Linear hashing is described by Litwin (1980). Dynamic hashing, which we did not discuss in detail, was proposed by Larson (1978). There are many proposed variations for extendible and linear hashing; for examples, see Cesarini and Soda (1991), Du and Tong (1991), and Hachem and Berra (1992).

Petitioner Uber Ex-1052, 0176

CHAPTER 6 | Index Structures for Files

In this chapter, we assume that a file already exists with some primary organization such as the unordered, ordered, or a hashed organizations that were described in Chapter 5. We will describe additional auxiliary **access structures** called **indexes**, which are used to speed up the retrieval of records in response to certain search conditions. The index structures typically provide **secondary access paths**, which provide alternative ways of accessing the records without affecting the physical placement of records on disk. They enable efficient access to records based on the **indexing fields** that are used to construct the index. Basically, *any field* of the file can be used to create an index and *multiple indexes* on different fields can be constructed on the same file. A variety of indexes are possible; each of them uses a particular data structure to speed up the search. To find a record or records in the file based on a certain selection criterion on an indexing field, one has to initially access the index, which points to one or more blocks in the file where the required records are located. The most prevalent types of indexes are based on ordered files (single-level indexes) and tree data structures (multilevel indexes, B^+-trees). Indexes can also be constructed based on hashing or other search data structures.

We describe different types of single-level ordered indexes—primary, secondary, and clustering—in Section 6.1. By viewing a single-level index as an ordered file, one can develop additional indexes for it, giving rise to the concept of multilevel indexes. A popular indexing scheme called ISAM (Indexed Sequential Access Method) is based on this idea. We discuss multilevel indexes in Section 6.2. In Section 6.3 we describe B-trees and B^+-trees, which are data structures that are commonly used in DBMSs to implement

Petitioner Uber Ex-1052, 0177

dynamically changing multilevel indexes. B^+-trees have become a commonly accepted default structure for generating indexes on demand in most relational DBMSs. Section 6.4 is devoted to the alternative ways of accessing data based on a combination of multiple keys. In Section 6.5, we discuss how other data structures—such as hashing—can be used to construct indexes. We also briefly introduce the concept of logical indexes, which give an additional level of indirection from physical indexes, allowing for the physical index to be flexible and extensible in its organization. Section 6.6 summarizes the chapter.

6.1 Types of Single-Level Ordered Indexes

The idea behind an ordered index access structure is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search an index to find a list of *addresses*—page numbers in this case—and use these addresses to locate a term in the textbook by *searching* the specified pages. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a linear search on a file. Of course, most books do have additional information, such as chapter and section titles, that can help us find a term without having to search through the whole book. However, the index is the only exact indication of where each term occurs in the book.

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field** (or **indexing attribute**).[1] The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are *ordered* so that we can do a binary search on the index. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient. Multilevel indexing (see Section 6.2) does away with the need for a binary search at the expense of creating indexes to the index itself.

There are several types of ordered indexes. A **primary index** is specified on the *ordering key field* of an ordered file of records. Recall from Section 5.8 that an ordering key field is used to *physically order* the file records on disk, and every record has a *unique value* for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field—another type of index, called a **clustering index**, can be used. Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, *but not both*. A third type of index, called a **secondary index**, can be specified on any *nonordering* field of a file. A file can have several secondary indexes in addition to its primary access method. In the next three subsections we discuss these three types of single-level indexes.

1. We will use the terms *field* and *attribute* interchangeably in this chapter.

Petitioner Uber Ex-1052, 0178

6.1.1 Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields. The first field is of the same data type as the ordering key field—called the **primary key**—of the data file, and the second field is a pointer to a disk block (a block address). There is one **index entry** (or **index record**) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values. We will refer to the two field values of index entry i as <K(i), P(i)>.

To create a primary index on the ordered file shown in Figure 5.9, we use the NAME field as primary key, because that is the ordering key field of the file (assuming that each value of NAME is unique). Each entry in the index has a NAME value and a pointer. The first three index entries are as follows:

<K(1) = (Aaron,Ed), P(1) = address of block 1>

<K(2) = (Adams,John), P(2) = address of block 2>

<K(3) = (Alexander,Ed), P(3) = address of block 3>

Figure 6.1 illustrates this primary index. The total number of entries in the index is the same as the *number of disk blocks* in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.[2]

Indexes can also be characterized as dense or sparse. A **dense index** has an index entry for *every search key value* (and hence every record) in the data file. A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values. A primary index is hence a nondense (sparse) index, since it includes an entry for each disk block of the data file rather than for every search value (or every record).

The index file for a primary index needs substantially fewer blocks than does the data file, for two reasons. First, there are *fewer index entries* than there are records in the data file. Second, each index entry is typically *smaller in size* than a data record because it has only two fields; consequently, more index entries than data records can fit in one block. A binary search on the index file hence requires fewer block accesses than a binary search on the data file.

A record whose primary key value is K lies in the block whose address is P(i), where $K(i) \leq K < K(i + 1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is P(i).[3]

2. We can use a scheme similar to the one described here, with the last record in each block (rather than the first) as the block anchor. This slightly improves the efficiency of the search algorithm.

3. Notice that the above formula would not be correct if the data file were ordered on a *nonkey field*; in that case the same index value in the block anchor could be repeated in the last records of the previous block.

Petitioner Uber Ex-1052, 0179

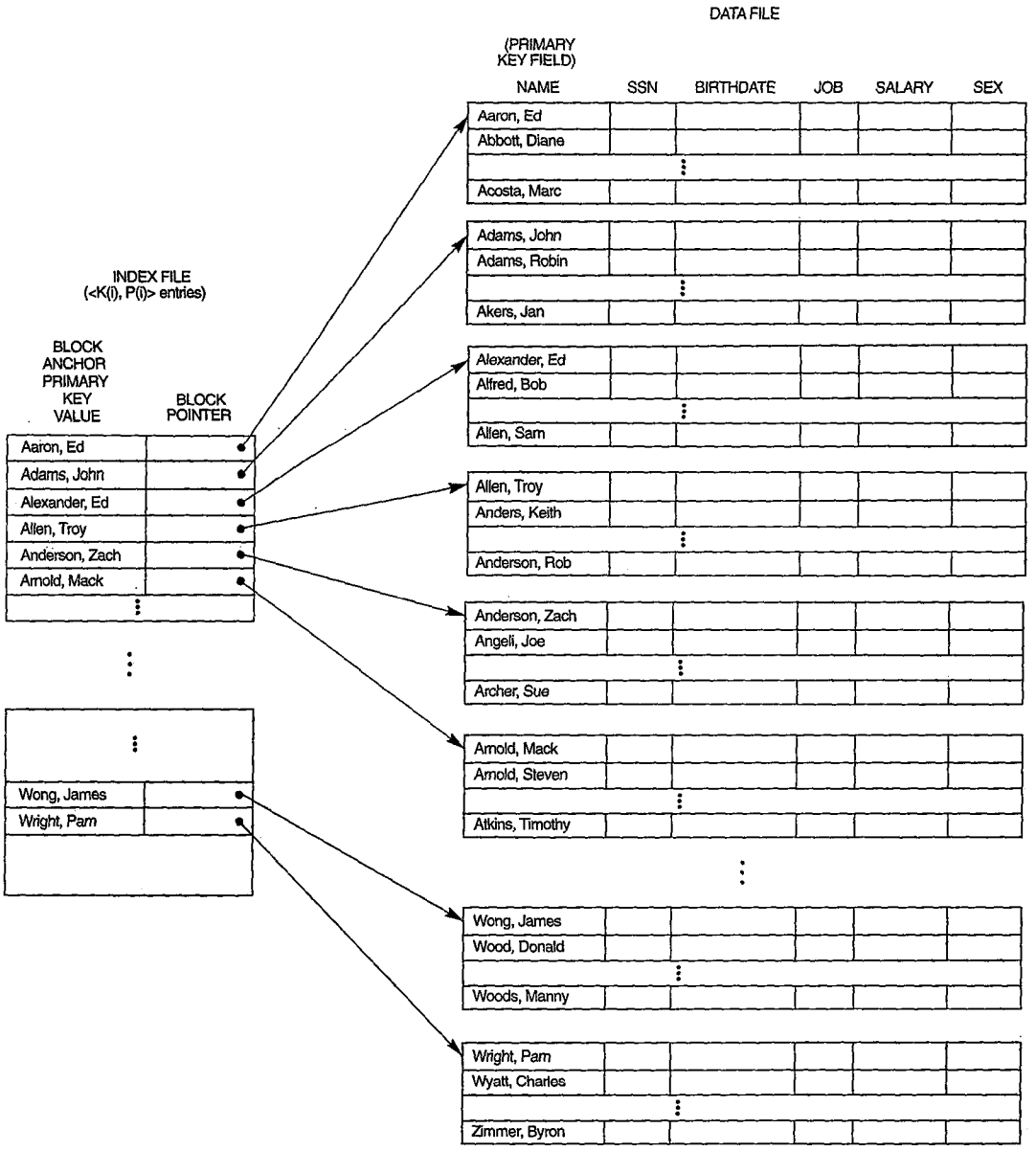


Figure 6.1 Primary index on the ordering key field of the file shown in Figure 5.9.

Petitioner Uber Ex-1052, 0180

Example 1 illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.

EXAMPLE 1: Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed size and are unspanned, with record length R = 100 bytes. The blocking factor for the file would be bfr = $\lfloor (B/R) \rfloor$ = $\lfloor (1024/100) \rfloor$ = 10 records per block. The number of blocks needed for the file is b = $\lceil (r/bfr) \rceil$ = $\lceil (30{,}000/10) \rceil$ = 3000 blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil$ = $\lceil (\log_2 3000) \rceil$ = 12 block accesses.

Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file. The size of each index entry is R_i = (9 + 6) = 15 bytes, so the blocking factor for the index is bfr_i = $\lfloor (B/R_i) \rfloor$ = $\lfloor (1024/15) \rfloor$ = 68 entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence b_i = $\lceil (r_i/bfr_i) \rceil$ = $\lceil (3000/68) \rceil$ = 45 blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil$ = $\lceil (\log_2 45) \rceil$ = 6 block accesses. To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses—an improvement over binary search on the data file, which required 12 block accesses.

A major problem with a primary index—as with any ordered file—is insertion and deletion of records. With a primary index, the problem is compounded because, if we attempt to insert a record in its correct position in the data file, we have to not only move records to make space for the new record but also change some index entries, since moving records will change the anchor records of some blocks. Using an unordered overflow file, as discussed in Section 5.8, can reduce this problem. Another possibility is to use a linked list of overflow records for each block in the data file. This is similar to the method of dealing with overflow records described with hashing in Section 5.9.2. Records within each block and its overflow linked list can be sorted to improve retrieval time. Record deletion is handled using deletion markers.

6.1.2 Clustering Indexes

If records of a file are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field**. We can create a different type of index, called a **clustering index**, to speed up retrieval of records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, containing the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 6.2 shows an example. Notice that record insertion and deletion still cause problems, because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous

Petitioner Uber Ex-1052, 0181

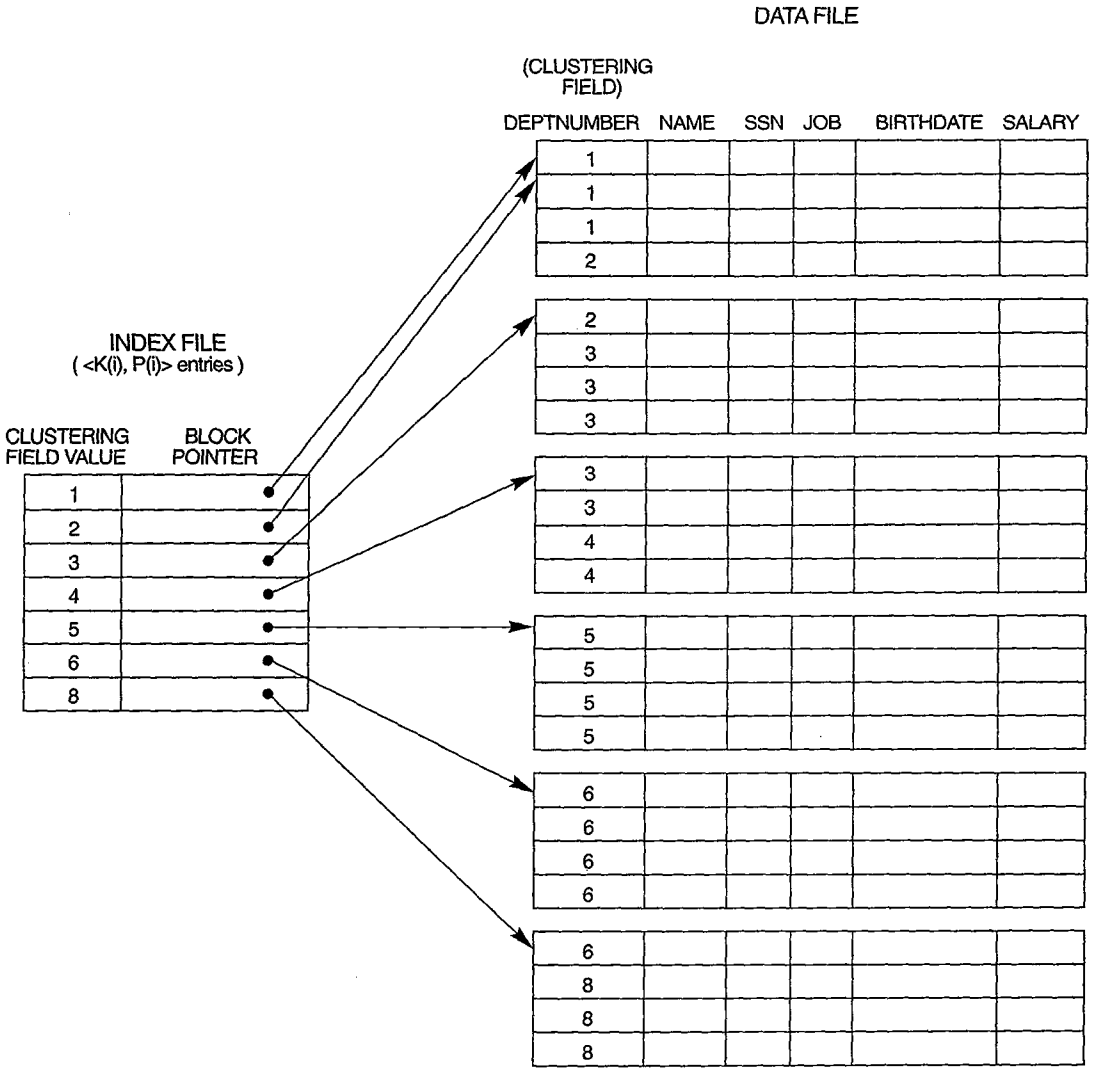


Figure 6.2 A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

blocks) for *each value* of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward. Figure 6.3 shows this scheme.

A clustering index is another example of a *nondense* index, because it has an entry for every *distinct value* of the indexing field rather than for every record in the file. There is some similarity between Figures 6.1 to 6.3, on the one hand, and Figure 5.13, on the other. An index is somewhat similar to the directory structures used for extendible hashing,

Petitioner Uber Ex-1052, 0182

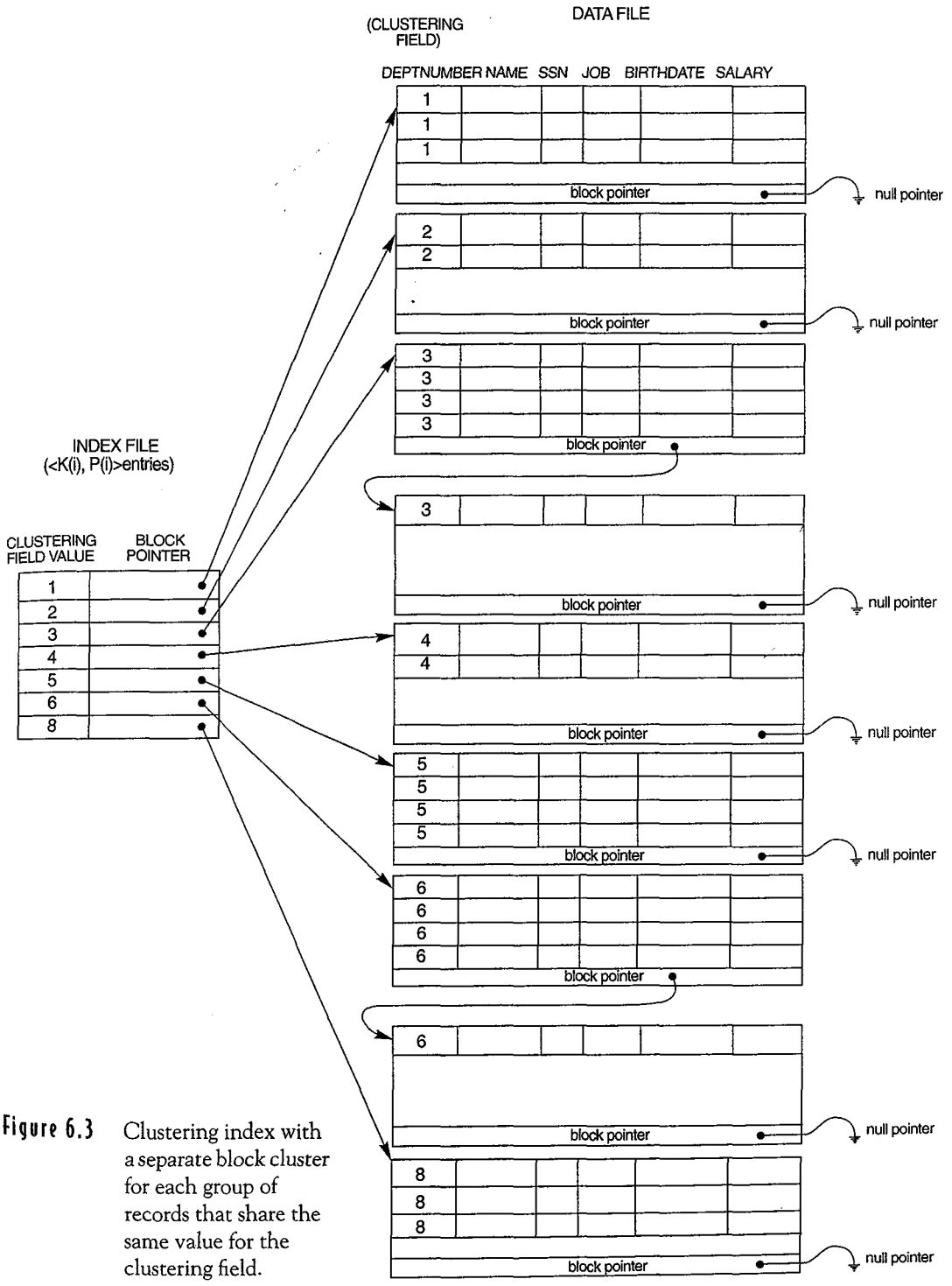


Figure 6.3 Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Petitioner Uber Ex-1052, 0183

described in Section 5.9.3. Both are searched to find a pointer to the data block containing the desired record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the hash value that is calculated by applying the hash function to the search field.

6.1.3 Secondary Indexes

A **secondary index** is also an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record* pointer. There can be *many* secondary indexes (and hence, indexing fields) for the same file.

We first consider a secondary index access structure on a key field that has a *distinct value* for every record. Such a field is sometimes called a **secondary key**. In this case there is one index entry for *each record* in the data file, which contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is **dense**.

We again refer to the two field values of index entry i as <K(i), P(i)>. The entries are **ordered** by value of K(i), so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we cannot use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index. Figure 6.4 illustrates a secondary index in which the pointers P(i) in the index entries are *block pointers*, not record pointers. Once the appropriate block is transferred to main memory, a search for the desired record within the block can be carried out.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a *linear search* on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist. Example 2 illustrates the improvement in number of blocks accessed.

EXAMPLE 2: Consider the file of Example 1 with r = 30,000 fixed-length records of size R = 100 bytes stored on a disk with block size B = 1024 bytes. The file has b = 3000 blocks, as calculated in Example 1. To do a linear search on the file, we would require b/2 = 3000/2 = 1500 block accesses on the average. Suppose that we construct a secondary index on a nonordering key field of the file that is V = 9 bytes long. As in Example 1, a block pointer is P = 6 bytes long, so each index entry is $R_i = (9 + 6) = 15$ bytes, and the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. In a dense secondary index such as this, the total number of index entries r_i is equal to the *number of records* in the data file, which is 30,000. The number of blocks needed for the index is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (30{,}000/68) \rceil = 442$ blocks.

Petitioner Uber Ex-1052, 0184

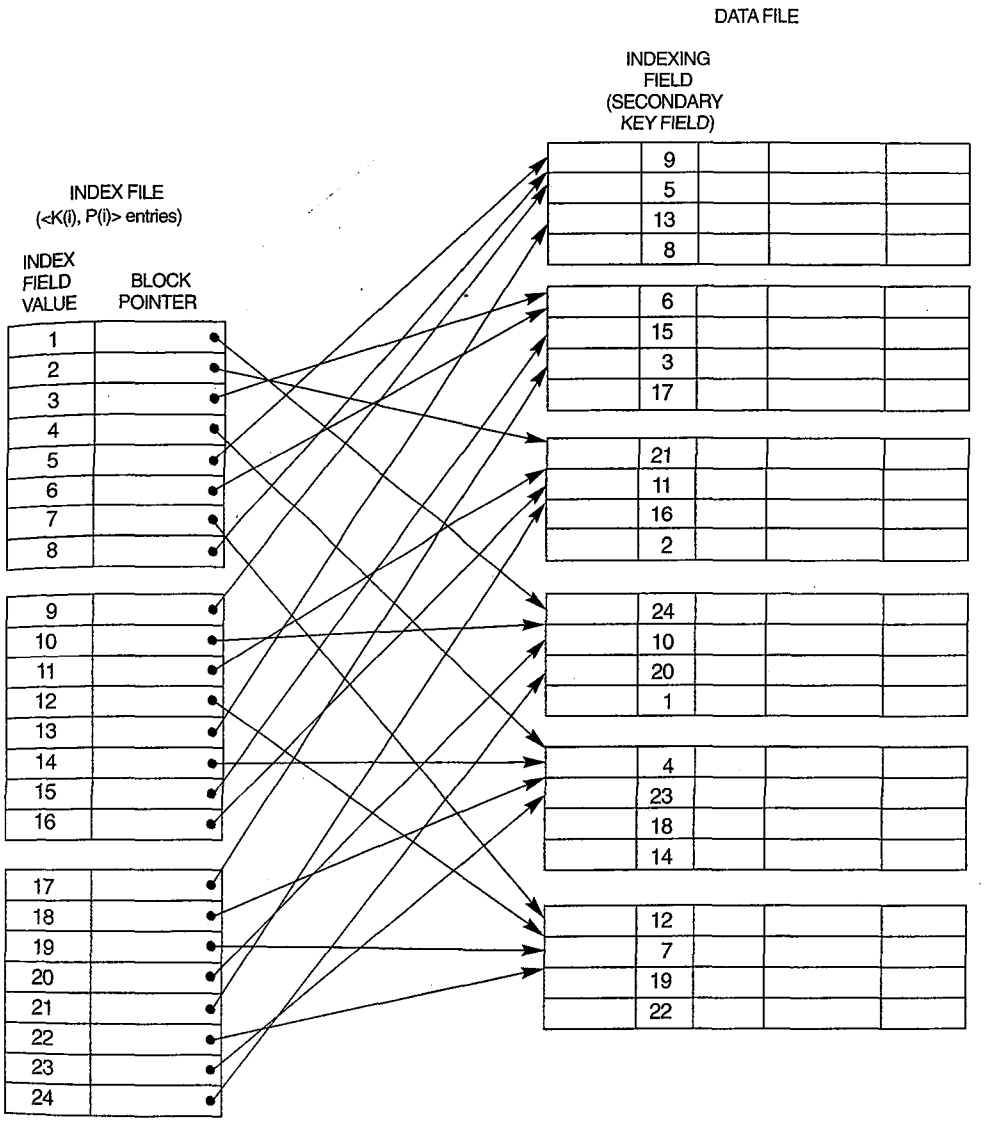


Figure 6.4 A dense secondary index (with block pointers) on a nonordering key field of a file.

Petitioner Uber Ex-1052, 0185

A binary search on this secondary index needs $\lceil(\log_2 b_i)\rceil = \lceil(\log_2 442)\rceil = 9$ block accesses. To search for a record using the index, we need an additional block access to the data file for a total of 9 + 1 = 10 block accesses—a vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the seven block accesses required for the primary index.

We can also create a secondary index on a *nonkey field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

- Option 1 is to include several index entries with the same K(i) value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers <P(i,1), ..., P(i,k)> in the index entry for K(i)—one pointer to each block that contains a record whose indexing field value equals K(i). In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each *index field value* but to create an extra level of indirection to handle the multiple pointers. In this nondense scheme, the pointer P(i) in index entry <K(i), P(i)> points to a *block of record pointers*; each record pointer in that block points to one of the data file records with value K(i) for the indexing field. If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 6.5. Retrieval via the index requires one or more additional block access because of the extra level, but the algorithms for searching the index and (more importantly) for inserting of new records in the data file are straightforward. In addition, retrievals on complex selection conditions may be handled by referring to the record pointers, without having to retrieve many unnecessary file records (see Exercise 6.19).

Notice that a secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field.

6.1.4 Summary

To conclude this section, we summarize the discussion on index types in two tables. Table 6.1 shows the index field characteristics of each type of ordered single-level index discussed—primary, clustering, and secondary. Table 6.2 summarizes the properties of each type of index by comparing the number of index entries and specifying which indexes are dense and which use block anchors of the data file.

Petitioner Uber Ex-1052, 0186

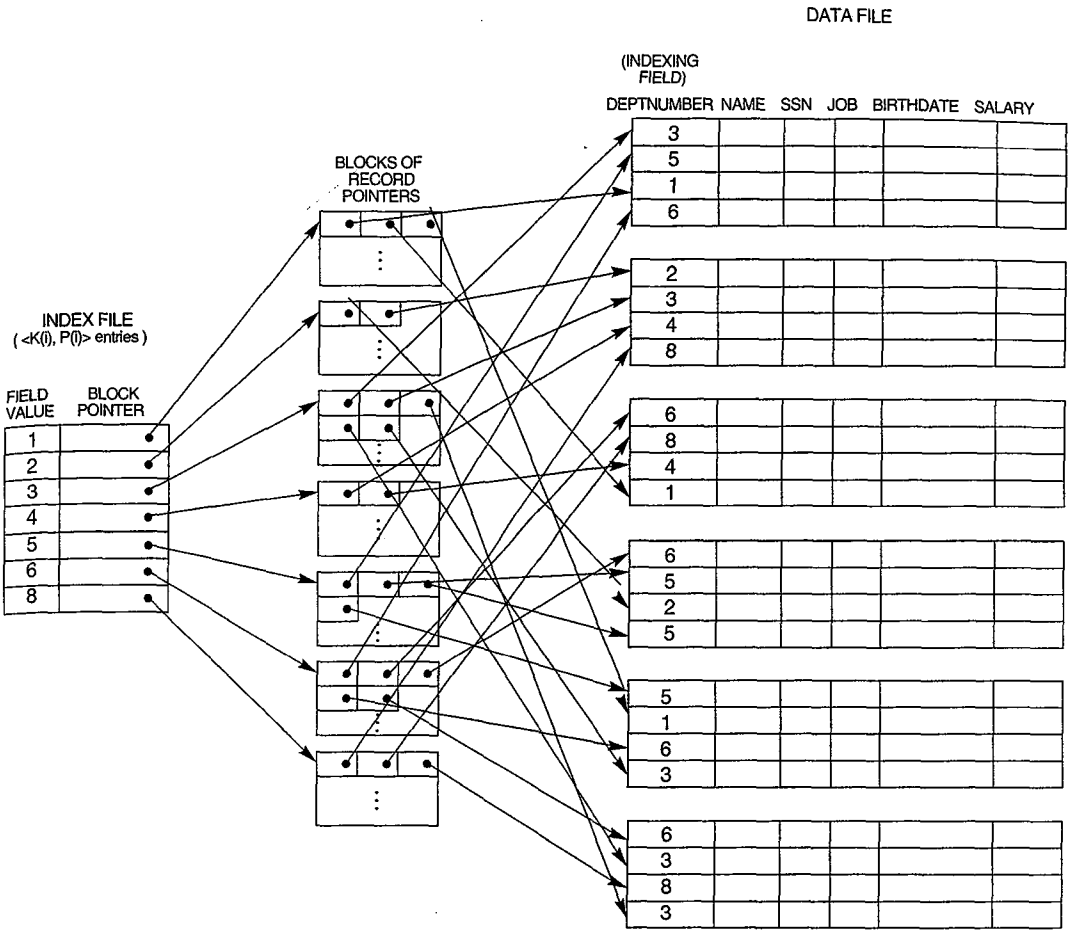


Figure 6.5 A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Table 6.1 Types of Indexes

	Ordering Field	Nonordering field
Key field	Primary index	Secondary index (key)
Nonkey field	Clustering index	Secondary index (nonkey)

Petitioner Uber Ex-1052, 0187

Table 6.2 Properties of Index Types

		Number of (First-level) Index Entries	Dense or Nondense	Block Anchoring on the Data File
Type of Index	**Primary**	Number of blocks in data file	Nondense	Yes
	Clustering	Number of distinct index field values	Nondense	Yes/no[a]
	Secondary (key)	Number of records in data file	Dense	No
	Secondary (nonkey)	Number of records[b] or Number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

6.2 Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value. A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks, because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why we take the log function to the base 2. The idea behind a **multilevel index** is to reduce the part of the index that we continue to search by bfr_i, the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfr_i is called the **fan-out** of the multilevel index, and we will refer to it by the symbol **fo**. Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a smaller number than for binary search if the fan-out is larger than 2.

A multilevel index considers the index file, which we will now refer to as the **first (or base) level** of a multilevel index, as an *ordered file* with a *distinct value* for each K(i). Hence we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for *each block* of the first level. The blocking factor bfr_i for the second level—and for all subsequent levels—is the same as that for the first-level index, because all index entries are the same size; each has one field value and one block address. If the first level has r_1 entries, and the blocking factor—which is also the fan-out—for the index is bfr_i = fo, then the first level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

We can repeat this process for the second level. The **third level**, which is a primary index for the second level, has an entry for each second-level block, so the number of

Petitioner Uber Ex-1052, 0188

third-level entries is $r_3 = \lceil (r_2/fo) \rceil$. Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the tth level is called the **top** index level.[4] Each level reduces the number of entries at the previous level by a factor of fo—the index fan-out—so we can use the formula $1 \leq (r_1/((fo)^t))$ to calculate t. Hence, a multilevel index with r_1 first-level entries will have approximately t levels, where $t = \lceil (\log_{fo}(r_1)) \rceil$.

The multilevel scheme described here can be used on any type of index, whether it is primary, clustering, or secondary—as long as the first-level index has *distinct values for K(i) and fixed-length entries*. Figure 6.6 shows a multilevel index built over a primary index. Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

EXAMPLE 3: Suppose that the dense secondary index of Example 2 is converted into a multilevel index. We calculated the index blocking factor $bfr_i = 68$ index entries per block, which is also the fan-out fo for the multilevel index; the number of first-level blocks $b_1 = 442$ blocks was also calculated. The number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ block. Hence, the third level is the top level of the index, and $t = 3$. To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses. Compare this to Example 2, where 10 block accesses were needed when a single-level index and binary search were used.

Notice that we could also have a multilevel primary index, which would be nondense. Exercise 6.14(c) illustrates this case, where we *must* access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first index level (without having to access a data block), since there is an index entry for *every* record in the file.

A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an **indexed sequential file** and was used in a large number of early IBM systems. Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is re-created during file reorganization. IBM's ISAM organization incorporates a two-level index that is closely related to the organization of the disk. The first level is a cylinder index, which has the key value of an anchor record for each cylinder of a disk pack and a pointer to the track index for the cylinder. The track index has the key value of an anchor record for each track in the cylinder and a pointer to the track. The track can then be searched sequentially for the desired record or block.

4. The numbering scheme for index levels used here is the reverse of the way levels are commonly defined for tree data structures. In tree data structures, t is referred to as level 0 (zero), $t - 1$ is level 1, etc.

Petitioner Uber Ex-1052, 0189

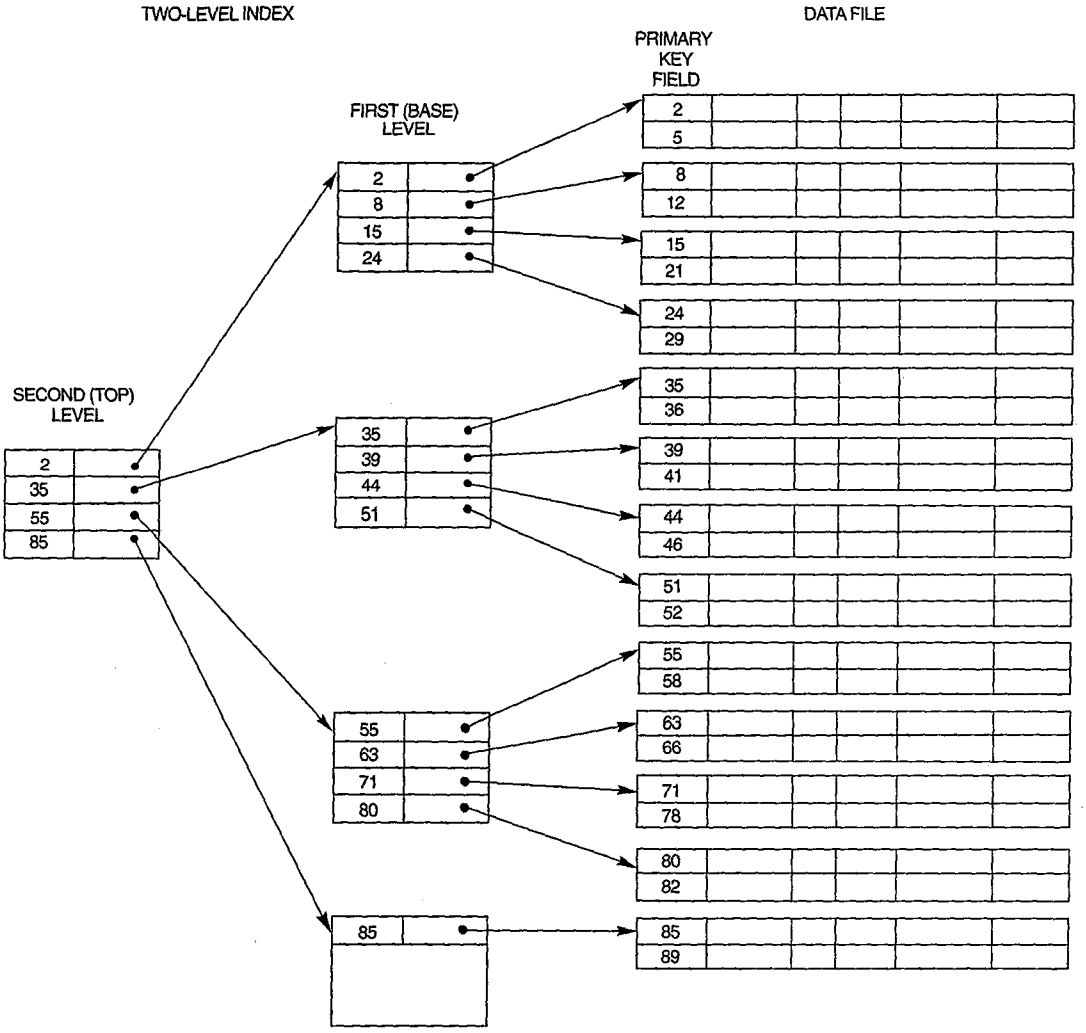


Figure 6.6 A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

Algorithm 6.1 outlines the search procedure for a record in a data file that uses a nondense multilevel primary index with t levels. We refer to entry i at level j of the index as $<K_j(i), P_j(i)>$, and we search for a record whose primary key value is K. We assume that any overflow records are ignored. If the record is in the file, there must be some entry at level 1 with $K_1(i) \leq K < K_1(i + 1)$ and the record will be in the block of the data file whose address is $P_1(i)$. Exercise 6.19 discusses modifying the search algorithm for other types of indexes.

Petitioner Uber Ex-1052, 0190

ALGORITHM 6.1 Searching a nondense multilevel primary index with t levels.

```
p ← address of top level block of index;
for j ← t step - 1 to 1 do
  begin
  read the index block (at j^th index level) whose address is p;
  search block p for entry i such that K_j(i) ≤ K < K_j(i + 1) (if K_j(i)
    is the last entry in the block, it is sufficient to satisfy K_j(i)
    ≤ K);
  p ← P_j(i) (* picks appropriate pointer at j^th index level *)
  end;
read the data file block whose address is p;
search block p for record with key = K;
```

As we have seen, a multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value. We are still faced with the problems of dealing with index insertions and deletions, because all index levels are *physically ordered files*. To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index that leaves some space in each of its blocks for inserting new entries. This is called a **dynamic multilevel index** and is often implemented by using data structures called B-trees and B^+-trees, which we describe in the next section.

6.3 Dynamic Multilevel Indexes Using B-Trees and B^+-Trees

B-trees and B^+-trees are special cases of the well-known tree data structure. We introduce very briefly the terminology used in discussing tree data structures. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and several—zero or more—**child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being zero.[5] A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node n and the subtrees of all the child nodes of n. Figure 6.7 illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes.

Usually, we display a tree with the root node at the top, as shown in Figure 6.7. One way to implement a tree is to have as many pointers in each node as there are child nodes of that node. In some cases, a parent pointer is also stored in each node. In addition to pointers, a node usually contains some kind of stored information. When a multilevel

5. This standard definition of the level of a tree node, which we use throughout Section 6.3, is different from the one we gave for multilevel indexes in Section 6.2.

Petitioner Uber Ex-1052, 0191

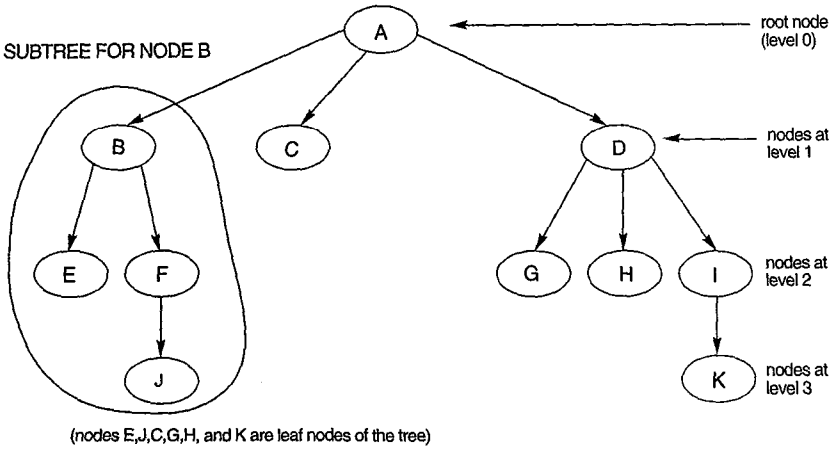


Figure 6.7 A tree data structure that shows an unbalanced tree.

index is implemented as a tree structure, this information includes the values of the file's indexing field that are used to guide the search for a particular record.

In Section 6.3.1, we introduce search trees and then discuss B-trees, which can be used as dynamic multilevel indexes to guide the search for records in a data file. B-tree nodes are kept between 50 and 100 percent full, and pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure. In Section 6.3.2 we discuss B^+-trees, a variation of B-trees in which pointers to the data blocks of a file are stored only in leaf nodes; this can lead to fewer levels and higher-capacity indexes.

6.3.1 Search Trees and B-Trees

A search tree is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. The multilevel indexes discussed in Section 6.2 can be thought of as a variation of a search tree; each node in the multilevel index can have as many as *fo* pointers and *fo* key values, where *fo* is the index fan-out. The index field values in each node guide us to the next node, until we reach the data file block that contains the required records. By following a pointer, we restrict our search at each level to a subtree of the search tree and ignore all nodes not in this subtree.

Search Trees. A search tree is slightly different from a multilevel index. A **search tree** of order p is a tree such that each node contains *at most* p − 1 search values and p pointers in the order $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$; each P_i is a pointer to a child node (or a null pointer); and each K_i is a search value from some ordered set of val-

Petitioner Uber Ex-1052, 0192

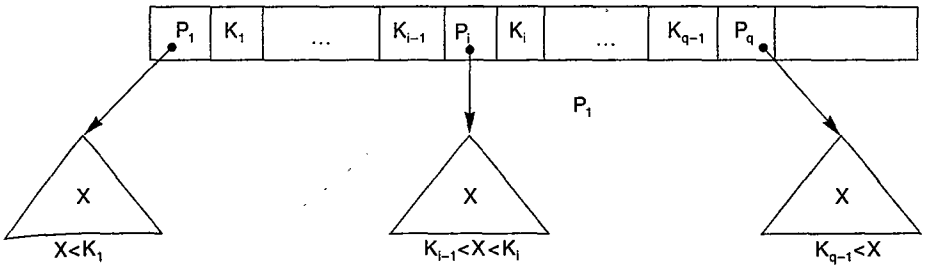


Figure 6.8 A node in a search tree with subtrees below it.

ues. All search values are assumed to be unique.[6] Figure 6.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < ... < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 6.8).

Whenever we search for a value X, we follow the appropriate pointer P_i according to the formulas in condition 2 above. Figure 6.9 illustrates a search tree of order $p = 3$ and integer search values. Notice that some of the pointers P_i in a node may be null pointers.

We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the **search field** (which is the same as the index field if a multilevel index guides the search). Each

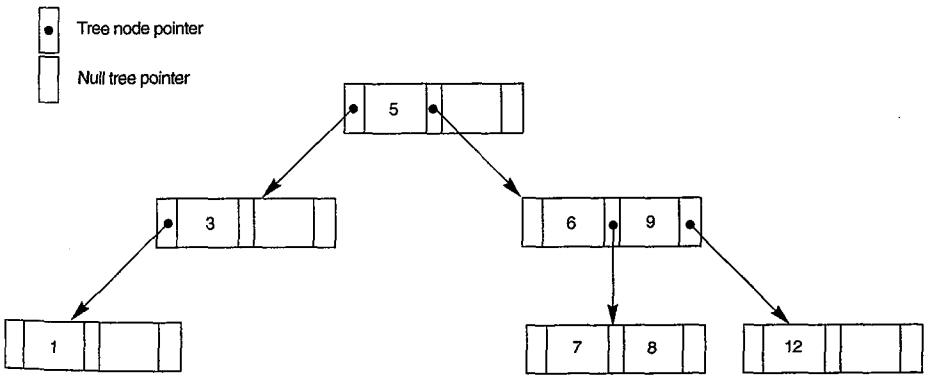


Figure 6.9 A search tree of order $p = 3$.

6. This restriction can be relaxed, but then the formulas that follow must be modified.

Petitioner Uber Ex-1052, 0193

key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is **balanced**, meaning that all of its leaf nodes are at the same level.[7] The tree in Figure 6.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. Keeping a search tree balanced is important because it guarantees that no nodes will be at very high levels and hence require many block accesses during a tree search. Another problem with search trees is that record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees. The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full. More formally, a **B-tree** of **order p,** when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure 6.10a) is of the form

   $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>$

   where $q \leq p$. Each P_i is a **tree pointer**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**[8]—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).
2. Within each node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search key field values X in the subtree pointed at by P_i (the ith subtree, see Figure 6.10a), we have:

   $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
4. Each node has at most p tree pointers.
5. Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.

7. The definition of *balanced* is different for binary trees. Balanced binary trees are known as AVL trees.

8. A data pointer is either a block address, or a record address; the latter is essentially a block address and a record offset within the block.

Petitioner Uber Ex-1052, 0194

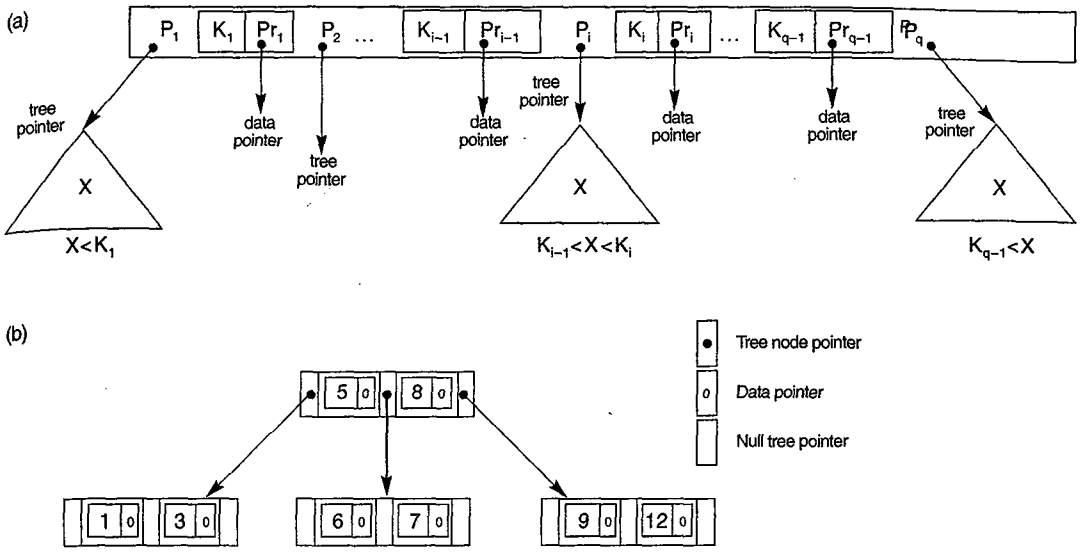


Figure 6.10 B-tree structures. (a) A node in a B-tree with q – 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

6. A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).
7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are null.

Figure 6.10(b) illustrates a B-tree of order p = 3. Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree *on a nonkey field*, we must change the definition of the file pointers Pr_i to point to a block—or cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to Option 3, discussed in Section 6.1.3, for secondary indexes.

A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with p – 1 search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split. We do not discuss algorithms for B-trees in detail here; rather, we outline search and insertion procedures for B^+-trees in the next section.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion

Petitioner Uber Ex-1052, 0195

can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69 percent full when the number of values in the tree stabilizes. This is also true of B^+-trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time. Example 4 illustrates how we calculate the order p of a B-tree stored on disk.

EXAMPLE 4: Suppose the search field is V = 9 bytes long, the disk block size is B = 512 bytes, a record (data) pointer is P_r = 7 bytes, and a block pointer is P = 6 bytes. Each B-tree node can have *at most* p tree pointers, p − 1 data pointers, and p − 1 search key field values (see Figure 6.10a). These must fit into a single disk block if each B-tree node is to correspond to a disk block. Hence, we must have:

$$(p * P) + ((p - 1) * (P_r + V)) \leq B$$
$$(p * 6) + ((p - 1) * (7 + 9)) \leq 512$$
$$(22 * p) \leq 528$$

We can choose p to be a large value that satisfies the above inequality, which gives p = 23 (p = 24 is not chosen because of the reasons given next).

In general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries q in the node and a pointer to the parent node. Hence, before we do the preceding calculation for p, we should reduce the block size by the amount of space needed for all such information. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

EXAMPLE 5: Suppose that the search field of Example 4 is a nonordering key field, and we construct a B-tree on this field. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have p * 0.69 = 23 * 0.69 or approximately 16 pointers and, hence, 15 search key field values. The **average fan-out** fo =16. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root:	1 node	15 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3840 entries	4096 pointers
Level 3:	4096 nodes	61,440 entries	

At each level, we calculated the number of entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size, pointer size, and search key field size, a two-level B-tree holds 3840 + 240 + 15 = 4095 entries on the average; a three-level B-tree holds 65,535 entries on the average.

B-trees are sometimes used as primary file organizations. In this case, whole records are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively *small number of records*, and a *small record*

Petitioner Uber Ex-1052, 0196

size. Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, B-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a B-tree of order p can have at most p−1 search values.

6.3.2 B^+-Trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B^+-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B^+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the B^+-tree are usually linked together to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B^+-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B^+-tree to guide the search. The structure of the *internal nodes* of a B^+-tree of order p (Figure 6.11a) is as follows:

1. Each internal node is of the form

   $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$

   where $q \leq p$ and each P_i is a **tree pointer**.
2. Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 6.11a).[9]
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the *leaf nodes* of a B^+-tree of order p (Figure 6.11b) is as follows:

1. Each leaf node is of the form

   $<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$

9. Our definition follows Knuth (1973). One can define a B^+-tree differently by exchanging the $<$ and $\leq$ symbols ($K_{i-1} \leq X < K_i$; $X < K_1$; $K_{q-1} \leq X$), but the principles remain the same.

Petitioner Uber Ex-1052, 0197

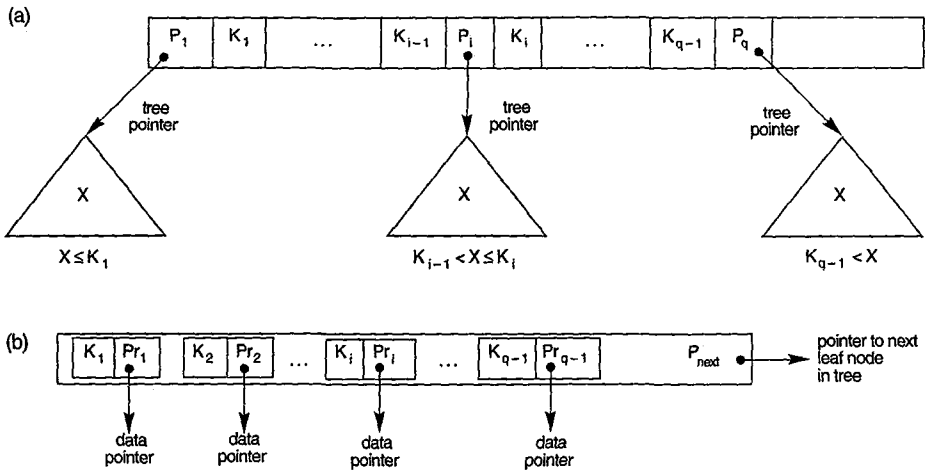


Figure 6.11 The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values. (b) Leaf node of a B^+-tree with $q-1$ search values and $q-1$ data pointers.

where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B^+-tree.

2. Within each leaf node, $K_1 < K_2 < ... < K_{q-1}$, $q \leq p$.
3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks—except for the P_{next} pointer, which is a tree pointer to the next leaf node. By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the P_{next} pointers. This provides ordered access to the data records on the indexing field. A $P_{previous}$ pointer can also be included. For a B^+-tree on a nonkey field, an extra level of indirection is needed similar to the one shown in Figure 6.5, so the Pr pointers are block pointers to blocks that contain a set of record pointers to the actual records in the data file, as discussed in Option 3 of Section 6.1.3.

Because entries in the *internal nodes* of a B^+-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a B^+-tree than for a similar B-tree. Thus, for the same block (node) size, the order p will be larger for the B^+-tree than for the B-tree, as we illustrate in Example 6. This can lead to fewer B^+-tree levels, improving search time. Because the structures for internal and for

Petitioner Uber Ex-1052, 0198

leaf nodes of a B^+-tree are different, the order p can be different. We will use p to denote the order for *internal nodes* and p_{leaf} to denote the order for *leaf nodes*, which we define as being the maximum number of data pointers in a leaf node.

EXAMPLE 6: To calculate the order p of a B^+-tree, suppose that the search key field is V = 9 bytes long, the block size is B = 512 bytes, a record pointer is P_r = 7 bytes, and a block pointer is P = 6 bytes, as in Example 4. An internal node of the B^+-tree can have up to p tree pointers and p − 1 search field values; these must fit into a single block. Hence, we have:

$$(p * P) + ((p - 1) * V) \leq B$$
$$(p * 6) + ((p - 1) * 9) \leq 512$$
$$(15 * p) \leq 521$$

We can choose p to be the largest value satisfying the above inequality, which gives p = 34. This is larger than the value of 23 for the B-tree, resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$$(p_{leaf} * (P_r + V)) + P \leq B$$
$$(p_{leaf} * (7 + 9)) + 6 \leq 512$$
$$(16 * p_{leaf}) \leq 506$$

It follows that each leaf node can hold up to p_{leaf} = 31 key value/data pointer combinations, assuming that the data pointers are record pointers.

As with the B-tree, we may need additional information—to implement the insertion and deletion algorithms—in each node. This information can include the type of node (internal or leaf), the number of current entries q in the node, and pointers to the parent and sibling nodes. Hence, before we do the above calculations for p and p_{leaf}, we should reduce the block size by the amount of space needed for all such information. The next example illustrates how we can calculate the number of entries in a B^+-tree.

EXAMPLE 7: Suppose that we construct a B^+-tree on the field of Example 6. To calculate the approximate number of entries of the B^+-tree, we assume that each node is 69 percent full. On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold 0.69 * p_{leaf} = 0.69 * 31 or approximately 21 data record pointers. A B^+-tree will have the following average number of entries at each level:

Root:	1 node	22 entries	23 pointers
Level 1:	23 nodes	506 entries	529 pointers
Level 2:	529 nodes	11,638 entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 record pointers	

Petitioner Uber Ex-1052, 0199

For the block size, pointer size, and search field size given above, a three-level B^+-tree holds up to 255,507 record pointers, on the average. Compare this to the 65,535 entries for the corresponding B-tree in Example 5.

Search, Insertion, and Deletion with B^+-Trees. Algorithm 6.2 outlines the procedure using the B^+-tree as access structure to search for a record. Algorithm 6.3 illustrates the procedure for inserting a record in a file with a B^+-tree access structure. These algorithms assume the existence of a key search field, and they must be modified appropriately for the case of a B^+-tree on a nonkey field. We now illustrate insertion and deletion with an example.

ALGORITHM 6.2 Searching for a record with search key field value K, using a B^+-tree.

```
n ← block containing root node of B+-tree;
read block n;
while (n is not a leaf node of the B+-tree) do
  begin
  q ← number of tree pointers in node n;
  if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
    then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
    else if K > n.Kq-1
      then n ← n.Pq
      else begin
        search node n for an entry i such that n.Ki-1 < K ≤ n.Ki;
        n ← n.Pi
        end;
  read block n
  end;
search block n for entry (Ki,Pri) with K = Ki; (* search leaf node *)
if found
  then read data file block with address Pri and retrieve record
  else record with search field value K is not in the data file;
```

ALGORITHM 6.3 Inserting a record with search key field value K in a B^+-tree of order p.

```
n ← block containing root node of B+-tree;
read block n; set stack S to empty;
while (n is not a leaf node of the B+-tree) do
  begin
  push address of n on stack S;
    (*stack S holds parent nodes that are needed in case of split*)
  q ← number of tree pointers in node n;
  if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
    then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
    else if K > n.Kq-1
```

Petitioner Uber Ex-1052, 0200

```
        then n ← n.Pq
        else begin
        search node n for an entry i such that n.Ki-1 < K ≤ n.Ki;
        n ← n.Pi
        end;
    read block n
    end;
search block n for entry (Ki,Pri) with K = Ki; (*search leaf node n*)
if found
    then record already in file-cannot insert
    else (*insert entry in B+-tree to point to record*)
    begin
    create entry (K,Pr) where Pr points to the new record;
    if leaf node n is not full
    then insert entry (K, Pr) in correct position in leaf node n
    else
    begin (*leaf node n is full with pleaf record pointers-is split*)
    copy n to temp (*temp is an oversize leaf node to hold extra entry*);
    insert entry (K, Pr) in temp in correct position;
      (*temp now holds pleaf + 1 entries of the form (Ki, Pri)*)
    new ← a new empty leaf node for the tree; new.Pnext ← n.Pnext;
    j ← ⌈(pleaf + 1)/2⌉;
    n ← first j entries in temp (up to entry (Kj,Prj)); n.Pnext ← new;
    new ← remaining entries in temp; K ← Kj;
(*now we must move (K,new) and insert in parent internal node
    -however, if parent is full, split may propagate*)
    finished ← false;
    repeat
    if stack S is empty
      then (*no parent node-new root node is created for the tree*)
      begin
      root ← a new empty internal node for the tree;
      root ← <n, K, new>; finished ← true;
    end
else
    begin
    n ← pop stack S;
    if internal node n is not full
      then
      begin (*parent node not full-no split*)
      insert (K, new) in correct position in internal node n;
      finished ← true
      end
      else
      begin (*internal node n is full with p tree pointers-is split*)
        copy n to temp (*temp is an oversize internal node*);
        insert (K,new) in temp in correct position;
          (*temp now has p+1 tree pointers*)
```

Petitioner Uber Ex-1052, 0201

```
            new ← a new empty internal node for the tree;
            j ← ⌊((p + 1)/2)⌋;
            n ← entries up to tree pointer Pj in temp;
              (*n contains <P1, K1, P2, K2, ..., Pj-1, Kj-1, Pj >*)
            new ← entries from tree pointer Pj+1 in temp;
              (*new contains < Pj+1, Kj+1, ..., Kp-1, Pp, Kp, Pp+1 >*)
            K ← Kj
          (*now we must move (K,new) and insert in parent internal node*)
            end
          end
        until finished
        end;
    end;
```

Figure 6.12 illustrates insertion of records in a B^+-tree of order p = 3 and p_{leaf} = 2. First, we observe that the root is the only node in the tree, so it is also a leaf node. As soon as more than one level is created, the tree is divided into internal nodes and leaf nodes. Notice that *every key value must exist at the leaf level*, because all data pointers are at the leaf level. However, only some values exist in internal nodes to guide the search. Notice also that every value appearing in an internal node also appears as *the rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

When a *leaf node* is full and a new entry is inserted there, the node **overflows** and must be split. The first $j = \lceil ((p_{leaf} + 1)/2) \rceil$ entries in the original node are kept there, and the remaining entries are moved to a new leaf node. The jth search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent. These must be inserted in the parent node in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split. The entries in the internal node up to P_j—the jth tree pointer after inserting the new value and pointer, where $j = \lfloor ((p + 1)/2) \rfloor$—are kept, while the jth search value is *moved* to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node (see Algorithm 6.3). This splitting can propagate all the way up to create a new root node and hence a new level for the B^+-tree.

Figure 6.13 illustrates deletion from a B^+-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node, because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case we try to find a **sibling** leaf node—a leaf node directly to the left or to the right of the node with underflow—and **redistribute** the entries among the node and its sibling so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try redistributing entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is not possible either, the three nodes are merged into two leaf nodes. In such a case, underflow may propagate to **internal** nodes because

Petitioner Uber Ex-1052, 0202

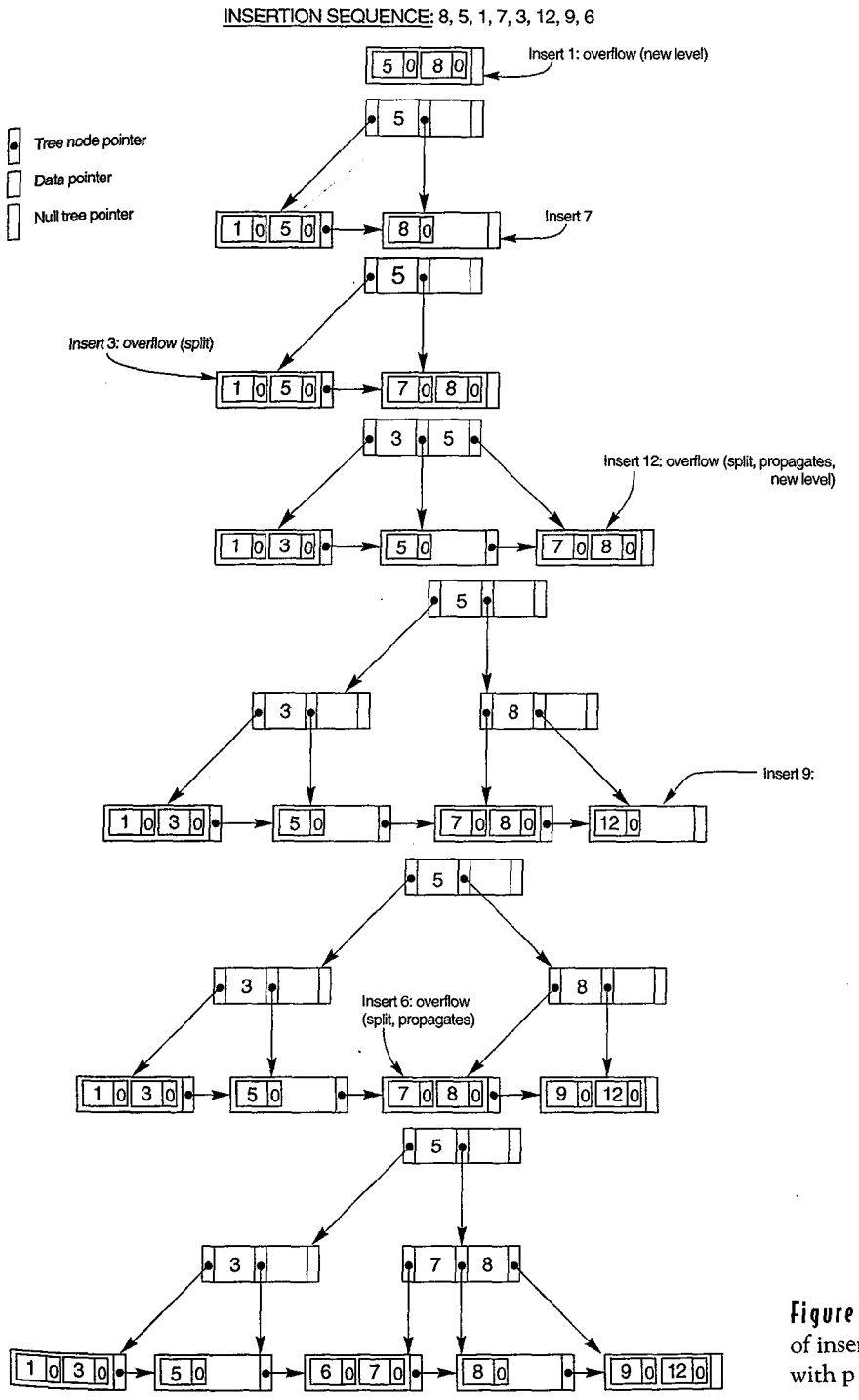


Figure 6.12 An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.

Petitioner Uber Ex-1052, 0203

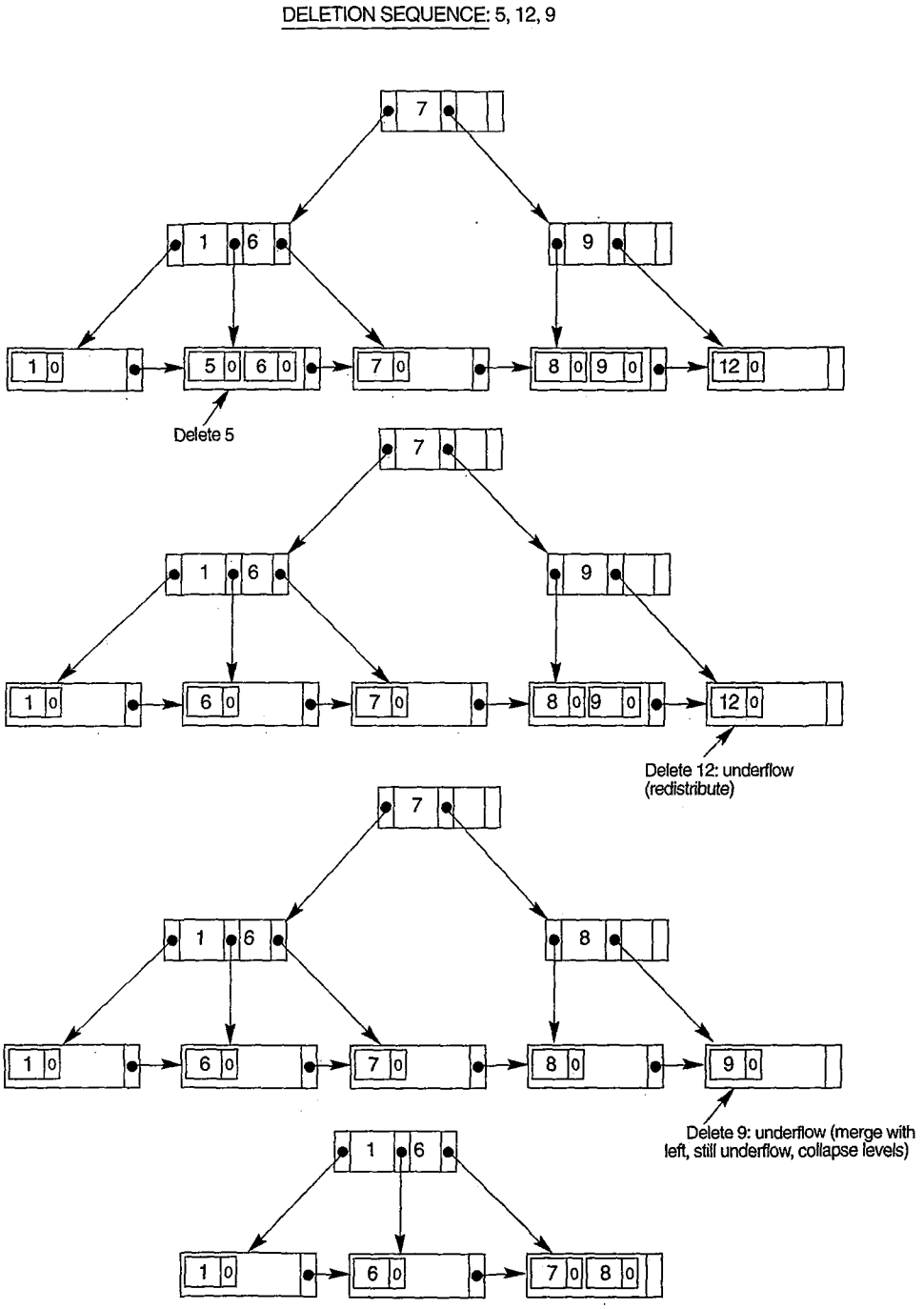


Figure 6.13 An example of deletion from a B^+-tree.

Petitioner Uber Ex-1052, 0204

one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 6.3. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.

Variations of B-Trees and B$^+$-Trees. To conclude this section, we briefly mention some variations of B-trees and B$^+$-trees. In some cases, constraint 5 on the B-tree (or B$^+$-tree), which requires each node to be at least half full, can be changed to require each node to be at least two-thirds full. In this case the B-tree has been called a **B*-tree**. In general, some systems allow the user to choose a **fill factor** between 0.5 and 1.0, where the latter means that the B-tree (index) nodes are to be completely full. It is also possible to specify two fill factors for a B$^+$-tree: one for the leaf level and one for the internal nodes of the tree. When the index is first constructed, each node is filled up to approximately the fill factors specified. Recently, investigators have suggested relaxing the requirement that a node be half full, and instead allow a node to become completely empty before merging, to simplify the deletion algorithm. Simulation studies show that this does not waste too much additional space under randomly distributed insertions and deletions.

6.4 Indexes on Multiple Keys

In our discussion so far, we assumed that the primary or secondary keys on which files were accessed were single attributes (fields). In many retrieval and update requests, multiple attributes are involved. If a certain combination of attributes is used very frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.

For example, consider an EMPLOYEE file containing attributes DNO (department number), AGE, STREET, CITY, ZIPCODE, SALARY and SKILL_CODE, with the key of SSN (social security number). Consider the query: "List the employees in department number 4 whose age is 59." Note that both DNO and AGE are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming DNO has an index, but AGE does not, access the records having DNO = 4 using the index then select from among them those records that satisfy AGE = 59.
2. Alternately, if AGE is indexed but DNO is not, access the records having AGE = 59 using the index then select from among them those records that satisfy DNO = 4 .
3. If indexes have been created on both DNO and AGE, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records). An intersection of these sets of records or pointers yields those records that satisfy both conditions.

Petitioner Uber Ex-1052, 0205

All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (DNO = 4 or AGE =59) individually are large, yet only a few records satisfy the combined condition, then none of the above is a very efficient technique for the given search request. A number of possibilities exist that would treat the combination <DNO, AGE>, or <AGE, DNO> as a search key made up of multiple attributes. We briefly outline these techniques below. We will refer to keys containing multiple attributes as **composite keys.**

6.4.1 Ordered Index on Multiple Attributes

All the discussion in this chapter so far applies if we create an index on a search key field that is a combination of <DNO, AGE>. The search key is a pair of values <4, 59> in the above example. In general, if an index is created on attributes $<A_1, A_2, ..., A_n>$, the search key values are tuples with n values: $<v_1, v_2, ..., v_n>$.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of department keys for department number 3 precede those for department 4. Thus <3, n> precedes <4, m> for any values of m and n. The ascending key order for keys with DNO = 4 would be <4, 18>, <4, 19>, <4, 20>, and so on. Lexicographic ordering works similarly to ordering of character strings. An index on a composite key of n attributes works similarly to any index discussed in this chapter so far.

6.4.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 5.9.2) that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of n components, the hash function is designed to produce a result with n separate hash addresses. The bucket address is a concatenation of these n addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite search key <DNO, AGE>. If DNO and AGE are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that DNO = 4 has a hash address "100" and AGE = 59 has hash address "10101". Then to search for the combined search value, DNO = 4 and AGE = 59, one goes to bucket address 100 10101; just to search for all employees with AGE = 59, all buckets (eight of them) will be searched whose addresses are "000 10101", "001 10101", ... etc. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be main-

Petitioner Uber Ex-1052, 0206

tained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

6.4.3 Grid Files

Another alternative is to organize the EMPLOYEE file as a grid file. If we want to access a file on two keys, say DNO and AGE as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure 6.14 shows a grid array for the EMPLOYEE file with one linear scale for DNO and another for the AGE attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for DNO has DNO = 1, 2 combined as one value 0 on the scale, while DNO = 5 corresponds to the value 2 on that scale. Similarly, AGE is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored. Figure 6.14 also shows assignment of cells to buckets (only partially).

Thus our request for DNO = 4 and AGE = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. Conceptually, the grid file concept may be applied to any number of search keys. For n search keys, the grid array would have n dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in terms of the grid array structure.

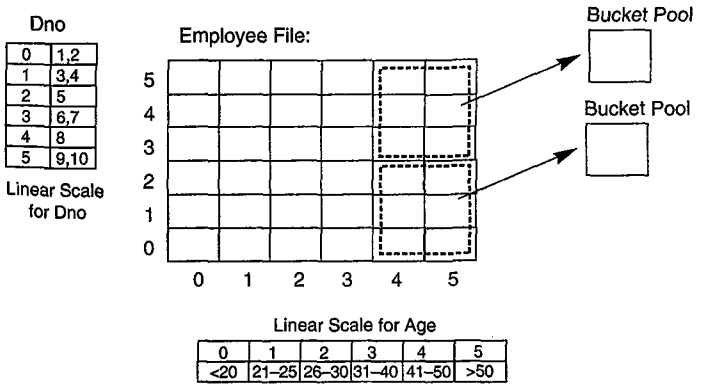


Figure 6.14 Example of a grid array on DNO and AGE attributes.

Petitioner Uber Ex-1052, 0207

Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.[10]

6.5 Other Types of Indexes

6.5.1 Using Hashing and Other Data Structures as Indexes

It is also possible to create access structures similar to indexes that are based on *hashing*. The index entries <K, Pr> (or <K, P>) can be organized as a dynamically expandable hash file, using one of the techniques described in Section 5.9.3; searching for an entry uses the hash search algorithm on K. Once an entry is found, the pointer Pr (or P) is used to locate the corresponding record in the data file. Other search structures can also be used as indexes.

6.5.2 Logical versus Physical Indexes

So far, we have assumed that the index entries <K, Pr> (or <K, P>) always include a physical pointer Pr (or P) that specifies the physical record address on disk as a block number and offset. This is sometimes called a **physical index**, and it has the disadvantage that the pointer must be changed if the record is moved to another disk location. For example, suppose that a primary file organization is based on linear hashing or extendible hashing; then, each time a bucket is split, some records are allocated to new buckets and hence have new physical addresses. If there was a secondary index on the file, the pointers to those records would have to be found and updated—a difficult task.

To remedy this situation, we can use a structure called a **logical index**, whose index entries are of the form <K, K_p>. Each entry has one value K for the secondary indexing field matched with the value K_p of the field used for the primary file organization. By searching the secondary index on the value of K, a program can locate the corresponding value of K_p and use this to access the record through the primary file organization. Logical indexes thus introduce an additional level of indirection between the access structure and the data. They are used when physical record addresses are expected to change frequently. The cost of this indirection is the extra search based on the primary file organization.

6.5.3 Discussion

In many systems, an index is not an integral part of the data file but can be created and discarded dynamically. That is why it is often called an *access structure*. Whenever we expect to access a file frequently based on some search condition involving a particular field, we can request the DBMS to create an index on that field. Usually, a secondary index is created to avoid physical ordering of the records in the data file on disk.

10. Insertion/deletion algorithms for grid files may be found in Nievergelt [1984].

Petitioner Uber Ex-1052, 0208

The main advantage of secondary indexes is that—theoretically, at least—they can be created in conjunction with *virtually any primary record organization*. Hence, a secondary index could be used to complement other primary access methods such as ordering or hashing, or it could even be used with mixed files. To create a B^+-tree secondary index on some field of a file, we must go through all records in the file to create the entries at the leaf level of the tree. These entries are then sorted and filled according to the specified fill factor; simultaneously, the other index levels are created. It is more expensive and much harder to create primary indexes and clustering indexes dynamically, because the records of the data file must be physically sorted on disk in order of the indexing field. However, some systems allow users to create these indexes dynamically on their files by sorting the file during index creation.

It is common to use an index to enforce a *key constraint* on an attribute. While searching the index to insert a new record, it is straightforward to check at the same time whether another record in the file—and hence in the index tree—has the same key attribute value as the new record. If so, the insertion can be rejected.

A file that has a secondary index on every one of its fields is often called a **fully inverted file**. Because all indexes are secondary, new records are inserted at the end of the file; therefore, the data file itself is an unordered (heap) file. The indexes are usually implemented as B^+-trees, so they are updated dynamically to reflect insertion or deletion of records. Some commercial DBMSs, such as ADABAS of Software-AG, use this method extensively.

We referred to the popular IBM file organization called ISAM in Section 6.2. Another IBM method, the **virtual storage access method** (VSAM), is somewhat similar to the B^+-tree access structure.

6.6 Summary

In this chapter we presented file organizations that involve additional access structures, called indexes, to improve the efficiency of retrieval of records from a data file. These access structures may be used *in conjunction with* the primary file organizations discussed in Chapter 5, which are used to organize the file records themselves on disk.

Three types of ordered single-level indexes were introduced: (1) primary, (2) clustering, and (3) secondary. Each index is specified on a field of the file. Primary and clustering indexes are constructed on the physical ordering field of a file, whereas secondary indexes are specified on non-ordering fields. The field for a primary index must also be a key of the file, whereas it is a non-key field for a clustering index. A single-level index is an ordered file and is searched using a binary search. We showed how multilevel indexes can be constructed to improve the efficiency of searching an index.

We then showed how multilevel indexes can be implemented as B-trees and B^+-trees, which are dynamic structures that allow an index to expand and shrink dynamically. The nodes (blocks) of these index structures are kept between half full and completely full by the insertion and deletion algorithms. Nodes eventually stabilize at an average occupancy of 69 percent full, allowing space for insertions without requiring reorganization of the index for the majority of insertions. B^+-trees can generally hold more

Petitioner Uber Ex-1052, 0209

entries in their internal nodes than can B-trees, so they may have fewer levels or hold more entries than does a corresponding B-tree.

We gave an overview of multiple key access methods, and showed how an index can be constructed based on hash data structures. We then introduced the concept of a logical index, and compared it with the physical indexes we described before. Finally, we discussed how combinations of the above organizations can be used. For example, secondary indexes are often used with mixed files, as well as with unordered and ordered files. Secondary indexes can also be created for hash files and dynamic hash files.

Review Questions

6.1. Define the following terms: *indexing field*, *primary key field*, *clustering field*, *secondary key field*, *block anchor*, *dense index*, and *non-dense (sparse) index*.

6.2. What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?

6.3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?

6.4. How does multilevel indexing improve the efficiency of searching an index file?

6.5. What is the order p of a B-tree? Describe the structure of B-tree nodes.

6.6. What is the order p of a B^+-tree? Describe the structure of both internal and leaf nodes of a B^+-tree.

6.7. How does a B-tree differ from a B^+-tree? Why is a B^+-tree usually preferred as an access structure to a data file?

6.8. Explain what alternative choices exist for accessing a file based on multiple search keys.

6.9. What is partitioned hashing? How does it work? What are its limitations?

6.10. What is a grid file? What are its advantages and disadvantages?

6.11. Show an example of constructing a grid array on two attributes on some file.

6.12. What is a fully inverted file? What is an indexed sequential file?

6.13. How can hashing be used to construct an index? What is the difference between a logical index and a physical index?

Exercises

6.14. Consider a disk with block size B = 512 bytes. A block pointer is P = 6 bytes long, and a record pointer is P_R = 7 bytes long. A file has r = 30,000 EMPLOYEE records of *fixed length*. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), DEPARTMENTCODE (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), JOBCODE (4 bytes), SALARY (4 bytes, real number). An additional byte is used as a deletion marker.

a. Calculate the record size R in bytes.

b. Calculate the blocking factor bfr and the number of file blocks b, assuming an unspanned organization.

Petitioner Uber Ex-1052, 0210

c. Suppose that the file is *ordered* by the key field SSN and we want to construct a *primary index* on SSN. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out fo); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its SSN value—using the primary index.

d. Suppose that the file is *not ordered* by the key field SSN and we want to construct a *secondary index* on SSN. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.

e. Suppose that the file is *not ordered* by the nonkey field DEPARTMENTCODE and we want to construct a *secondary index* on DEPARTMENTCODE, using option 3 of Section 6.1.3, with an extra level of indirection that stores record pointers. Assume there are 1000 distinct values of DEPARTMENTCODE and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out fo); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it into a multilevel index; (v) the total number of blocks required by the multilevel index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file that have a specific DEPARTMENTCODE value, using the index.

f. Suppose that the file is *ordered* by the nonkey field DEPARTMENTCODE and we want to construct a *clustering index* on DEPARTMENTCODE that uses block anchors (every new value of DEPARTMENTCODE starts at the beginning of a new block). Assume there are 1000 distinct values of DEPARTMENTCODE and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out fo); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve all records in the file that have a specific DEPARTMENTCODE value, using the clustering index (assume that multiple blocks in a cluster are contiguous).

g. Suppose that the file is *not* ordered by the key field SSN and we want to construct a B^+-tree access structure (index) on SSN. Calculate (i) the orders p and p_{leaf} of the B^+-tree; (ii) the number of leaf-level blocks needed if blocks are approximately 69 percent full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69 percent full (rounded up for convenience); (iv) the total number of blocks required by the B^+-tree; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its SSN value—using the B^+-tree.

h. Repeat part g, but for a B-tree rather than for a B^+-tree. Compare your results for the B-tree and for the B^+-tree.

Petitioner Uber Ex-1052, 0211

6.15. A PARTS file with Part# as key field includes records with the following Part# values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose that the search field values are inserted in the given order in a B^+-tree of order p = 4 and p_{leaf} = 3; show how the tree will expand and what the final tree will look like.

6.16. Repeat Exercise 6.15, but use a B-tree of order p = 4 instead of a B^+-tree.

6.17. Suppose that the following search field values are deleted, in the given order, from the B^+-tree of Exercise 6.15; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

6.18. Repeat Exercise 6.17, but for the B-tree of Exercise 6.16.

6.19. Algorithm 6.1 outlines the procedure for searching a nondense multilevel primary index to retrieve a file record. Adapt the algorithm for each of the following cases:
   a. A multilevel secondary index on a nonkey nonordering field of a file. Assume that option 3 of Section 6.1.3 is used, where an extra level of indirection stores pointers to the individual records with the corresponding index field value.
   b. A multilevel secondary index on a nonordering key field of a file.
   c. A multilevel clustering index on a nonkey ordering field of a file.

6.20. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 6.1.3; for example, we could have secondary indexes on the fields DEPARTMENTCODE, JOBCODE, and SALARY of the EMPLOYEE file of Exercise 6.14. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (DEPARTMENTCODE = 5 AND JOBCODE <= 12 AND SALARY = 50,000), using the record pointers in the indirection level.

6.21. Adapt Algorithms 6.2 and 6.3, which outline search and insertion procedures for a B^+-tree, to a B-tree.

6.22. It is possible to modify the B^+-tree insertion algorithm to delay the case where a new level is produced by checking for a possible *redistribution* of values among the leaf nodes. Figure 6.15 illustrates how this could be done for our example in Figure 6.12; rather than splitting the leftmost leaf node when 12 is inserted, we do a *left redistribution* by moving 7 to the leaf node to its left (if there is space in this node). Figure 6.15 shows how the tree would look when redistribution is considered. It is also possible to consider *right redistribution*. Try to modify the B^+-tree insertion algorithm to take redistribution into account.

6.23. Outline an algorithm for deletion from a B^+-tree.

6.24. Repeat Exercise 6.23 for a B-tree.

Selected Bibliography

Bayer and McCreight (1972) introduced B-trees and associated algorithms. Comer (1979) provides an excellent survey of B-trees and their history, and variations of B-trees. Knuth (1973) provides detailed analysis of many search techniques, including B-trees and some of their variations. Nievergelt (1974) discusses the use of binary search trees for file organization. Textbooks on file structures including Wirth (1972), Claybrook (1983),

Petitioner Uber Ex-1052, 0212

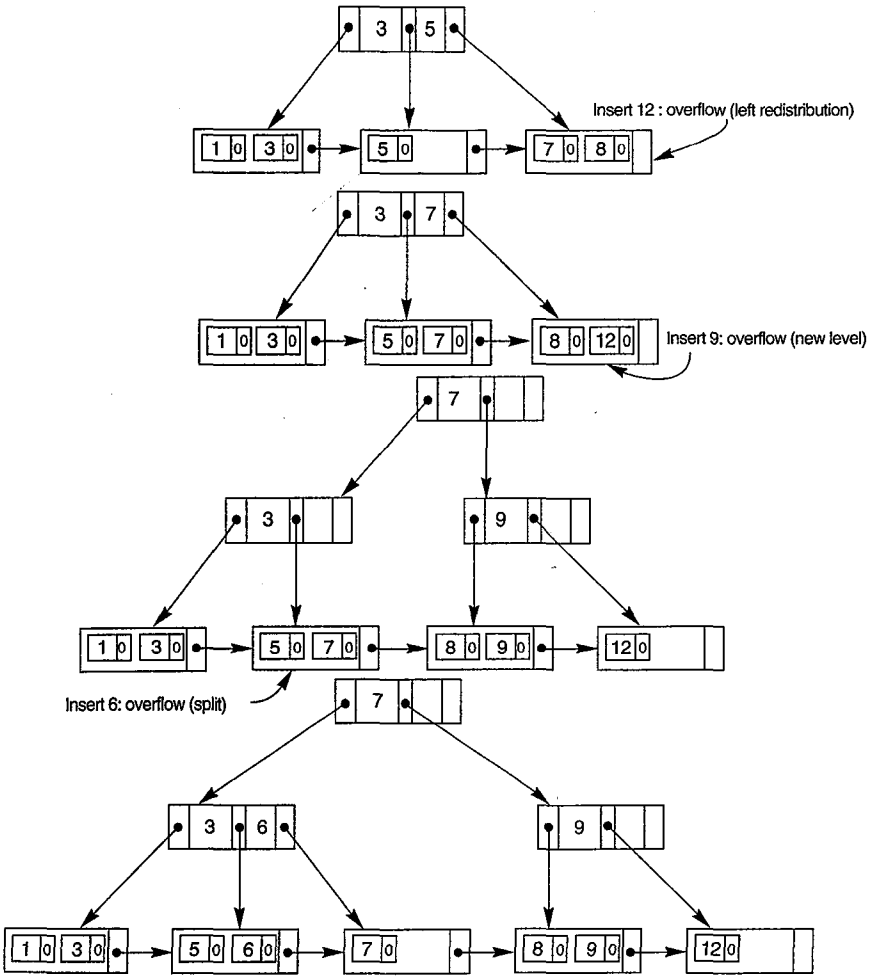


Figure 6.15 B^+-tree insertion with left redistribution.

Smith and Barnes (1987), Miller (1987), and Salzberg (1988) discuss indexing in detail and may be consulted for search, insertion, and deletion algorithms for B-trees and B^+-trees. Larson (1981) analyzes index-sequential files, and Held and Stonebraker (1978) compares static multilevel indexes with B-tree dynamic indexes. Lehman and Yao (1981) and Srinivasan and Carey (1991) did further analysis of concurrent access to B-trees. The books by Wiederhold (1983), Smith and Barnes (1987), and Salzberg (1988) among others, discuss many of the search techniques described in this chapter. Grid files are introduced in Nievergelt (1984). Partial-match retrieval, which uses partitioned hashing, is discussed in Burkhard (1976, 1979).

Petitioner Uber Ex-1052, 0213'

New techniques and applications of indexes and B^+-trees are discussed in Lanka and Mays (1991), Zobel et al. (1992), and Faloutsos and Jagadish (1992). Mohan and Narang (1992) discuss index creation. The performance of various B-tree and B^+-tree algorithms is assessed in Baeza-Yates and Larson (1989) and Johnson and Shasha (1993). Buffer management for indexes is discussed in Chan et al. (1992).

Petitioner Uber Ex-1052, 0214

PART 2

Relational Model, Languages, and Systems

CHAPTER 7 The Relational Data Model, Relational Constraints, and the Relational Algebra

This chapter opens Part II of the book on relational databases. The relational model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper [Codd 1970], and attracted immediate attention due to its simplicity and mathematical foundations. The model uses the concept of a *mathematical relation*—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first order predicate logic. In this chapter we discuss the basic characteristics of the model, its constraints, and the relational algebra, which is a set of operations for the relational model. The model has been implemented in a large number of commercial systems over the last twenty or so years.

Because of the amount of material related to the relational model, we have devoted the whole of Part II of this textbook to it. In Chapter 8, we will describe the SQL query language, which is the *standard* for commercial relational DBMSs. Chapter 9 presents additional topics concerning relational databases. Sections 9.1 and 9.2 present algorithms for designing a relational database schema by mapping a conceptual schema in the ER or EER model (see Chapters 3 and 4) into a relational representation. These mappings are incorporated into many database design and CASE[1] tools. The remainder of Chapter 9 presents some other relational languages. Chapter 10 presents an overview of two commercial relational DBMSs—ORACLE and Microsoft ACCESS. Chapters 14 and 15 in Part IV of the

1. CASE stands for Computer Aided Software Engineering.

Petitioner Uber Ex-1052, 0216

book present another aspect of the relational model, namely the formal constraints of functional and multivalued dependencies; these dependencies are used to develop a relational database design theory based on the concept known as *normalization*.

Data models that preceded the relational model include the hierarchical and network models. They were proposed in the sixties and were implemented in early DBMSs during the seventies and eighties. Because of their historical importance and the large existing user base for these DBMSs, we have included a summary of the highlights of these models in Appendices C and D. These models and systems will be with us for many years and are today being called *legacy systems*.

In this chapter, we will concentrate on describing the basic principles of the relational model of data. We begin by defining the modeling concepts and notation of the relational model in Section 7.1. Section 7.2 is devoted to a discussion of relational constraints that are now considered an important part of the relational model and are automatically enforced in most relational DBMSs. Section 7.3 defines the update operations of the relational model and discusses how violations of integrity constraints are handled.

In Section 7.4 we present a detailed discussion of the relational algebra, which is a collection of operations for manipulating relations and specifying queries. The relational algebra is an integral part of the relational data model. Section 7.5 defines additional relational operations that were added to the basic relational algebra because of their importance to many database applications. We give examples of specifying queries that use relational operations in Section 7.6. The same queries are used in subsequent chapters to illustrate various languages. Section 7.7 summarizes the chapter.

For the reader who is interested in a less detailed introduction to relational concepts, Sections 7.1.2, 7.4.7, and 7.5 may be skipped.

7.1 Relational Model Concepts

The relational model represents the database as a collection of *relations*. Informally, each relation resembles a table of values or, to some extent, a "flat" file of records. For example, the database of files that was shown in Figure 1.2 is considered to be in the relational model. However, there are important differences between relations and files, as we shall soon see.

When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. We introduced entity types and relationship types as concepts for modeling real-world data in Chapter 3. In the relational model, each row in the table represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help in interpreting the meaning of the values in each row. For example, the first table of Figure 1.2 is called STUDENT because each row represents facts about a particular student entity. The column names—Name, StudentNumber, Class, Major—specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of

Petitioner Uber Ex-1052, 0217

values that can appear in each column is called a *domain*. We now define these terms—*domain*, *tuple*, *attribute*, and *relation*—more precisely.

7.1.1 Domains, Attributes, Tuples, and Relations

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:

- USA_phone_numbers: The set of 10-digit phone numbers valid in the United States.
- Local_phone_numbers: The set of 7-digit phone numbers valid within a particular area code in the United States.
- Social_security_numbers: The set of valid 9-digit social security numbers.
- Names: The set of names of persons.
- Grade_point_averages: Possible values of computed grade point averages; each must be a real (floating point) number between 0 and 4.
- Employee_ages: Possible ages of employees of a company; each must be a value between 15 and 80 years old.
- Academic_department_names: The set of academic department names, such as Computer Science, Economics, and Physics, in a university.
- Academic_department_codes: The set of academic department codes, such as CS, ECON, and PHYS, in a university.

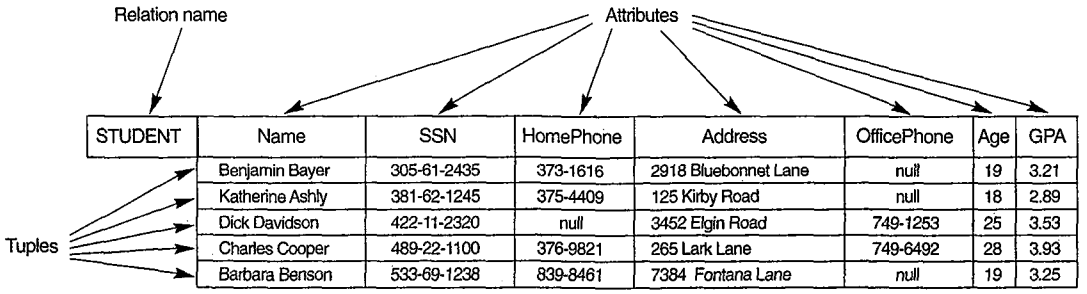
The preceding are called *logical* definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain USA_phone_numbers can be declared as a character string of the form (ddd)ddd-dddd, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80. For Academic_department_names, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as Person_weights should have the units of measurement—pounds or kilograms.

A **relation schema** R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes $A_1, A_2, \ldots, A_n$. Each **attribute** A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by $dom(A_i)$. A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** of a relation is the number of attributes n of its relation schema.

An example of a relation schema for a relation of degree 7, which describes university students, is the following:

STUDENT(Name, SSN, HomePhone, Address, OfficePhone, Age, GPA)

Petitioner Uber Ex-1052, 0218



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

Figure 7.1 The attributes and tuples of a relation STUDENT.

For this relation schema, STUDENT is the name of the relation, which has seven attributes. We can specify the following previously defined domains for some of the attributes of the STUDENT relation: dom(Name) = Names; dom(SSN) = Social_security_numbers; dom(HomePhone) = Local_phone_numbers, dom(OfficePhone) = Local_phone_numbers, and dom(GPA) = Grade_point_averages.

A **relation** (or **relation state**)[2] r of the relation schema $R(A_1, A_2, \ldots, A_n)$, also denoted by r(R), is a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$. Each **n-tuple** t is an ordered list of n values $t = <v_1, v_2, \ldots, v_n>$, where each value v_i, $1 \leq i \leq n$, is an element of $dom(A_i)$ or is a special **null** value. The ith value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$. The terms relation **intension** for the schema R and relation **extension** for a relation state r(R) are also commonly used.

Figure 7.1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema specified above. Each tuple in the relation represents a particular student entity. We display the relation as a table, where each tuple is shown as a row and each attribute corresponds to a column header indicating a role or interpretation of the values in that column. *Null values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuples.

The above definition of a relation can be *restated* as follows. A relation r(R) is a **mathematical relation** of degree n on the domains $dom(A_1)$, $dom(A_2)$, . . ., $dom(A_n)$, which is a **subset** of the **Cartesian product** of the domains that define R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n))$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the number of values or **cardinality** of a domain D by $|D|$, and assume that all domains are finite, the total number of tuples in the Cartesian product is:

$$|dom(A_1)| * |dom(A_2)| * \ldots * |dom(A_n)|$$

2. This has also been called a **relation instance.** We will not use this term because *instance* is also used to refer to a single tuple or row.

Petitioner Uber Ex-1052, 0219

Out of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real world changes, so does the relation, by being transformed into another relation state. However, the schema R is relatively static and does *not* change except very infrequently—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

It is possible for several attributes to *have the same domain*. The attributes indicate different **roles**, or interpretations, for the domain. For example, in the STUDENT relation, the same domain Local_phone_numbers plays the role of HomePhone, referring to the "home phone of a student," and the role of OfficePhone, referring to the "office phone of the student."

7.1.2 Characteristics of Relations

The earlier definition of relations implies certain characteristics that make a relation different from a file or a table. We now discuss some of these characteristics.

Ordering of Tuples in a Relation. A *relation* is defined as a set of tuples. Mathematically, elements of a set have *no order* among them; hence tuples in a relation do not have any particular order. However, in a file, records are physically stored on disk so there always is an order among the records. This ordering indicates first, second, i^{th}, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is not part of a relation definition, because a relation attempts to represent facts at a logical or abstract level. Many logical orders can be specified on a relation; for example, tuples in the STUDENT relation in Figure 7.1 could be logically ordered by values of Name, SSN, Age, or some other attribute. The definition of a relation does not specify any order: there is *no preference* for one logical ordering over another. Hence, the relation displayed in Figure 7.2 is considered *identical* to the one shown in Figure 7.1. When a relation is implemented as a file, a physical ordering may be specified on the records of the file.

Ordering of Values within a Tuple, and an Alternative Definition of a Relation. According to the preceding definition of a relation, an n-tuple is an *ordered list* of n values, so the ordering of values in a tuple—and hence of attributes in a relation schema

STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21

Figure 7.2 The relation STUDENT from Figure 7.1, with a different order of tuples.

Petitioner Uber Ex-1052, 0220

t = < (Name, Dick Davidson),(SSN, 422-11-2320),(HomePhone, null),(Address, 3452 Elgin Road),
(OfficePhone, 749-1253),(Age, 25),(GPA, 3.53)>

t = < (Address, 3452 Elgin Road),(Name, Dick Davidson),(SSN, 422-11-2320),(Age, 25),
(OfficePhone,749-1253),(GPA, 3.53),(HomePhone, null)>

Figure 7.3 Two identical tuples when order of attributes and values is not part of the definition of a relation.

definition—*is* important. However, at a logical level, the order of attributes and their values are *not* really important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary*. In this definition, a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a *set* of attributes, and a relation $r(R)$ is a finite set of **mappings** $r = \{t_1, t_2, \ldots, t_m\}$, where each tuple t_i is a mapping from R to D, and D is the union of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \ldots \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r. Each mapping t_i is called a tuple.

According to this definition, a **tuple** can be considered as a set of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is *not* important, because the attribute name appears with its value. By this definition, the two tuples shown in Figure 7.3 are identical. This makes sense at an abstract or logical level, since there really is no reason to prefer having one attribute value appear before another in a tuple.

When a relation is implemented as a file, the attributes are physically ordered as fields within a record. We will use the **first definition** of relation, where the attributes and the values within tuples *are ordered*, because it simplifies much of the notation. However, the alternative definition given here is more general.

Values in the Tuples. Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes (see Chapter 3) are not allowed. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.[3] Multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes. Recent research in the relational model attempts to remove these restrictions by using the concept of **nonfirst normal form** or **nested** relations (see Chapter 13).

The values of some attributes within a particular tuple may be unknown or may not apply to that tuple. A special value, called **null**, is used for these cases. For example, in Figure 7.1, some student tuples have null for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a

3. We discuss this assumption in more detail in Chapter 14.

Petitioner Uber Ex-1052, 0221

null for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*). In general, we can have *several types* of null values, such as "value unknown," "value exists but not available," or "attribute does not apply to this tuple." It is possible to devise different codes for different types of null values. Incorporating different types of null values into relational model operations has proved difficult, and a full discussion is outside the scope of this book.

Interpretation of a Relation. The relation schema can be interpreted as a declaration or a type of **assertion.** For example, the schema of the STUDENT relation of Figure 7.1 asserts that, in general, a student entity has a Name, SSN, HomePhone, Address, OfficePhone, Age, and GPA. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. For example, the first tuple in Figure 7.1 asserts the fact that there is a STUDENT whose name is Benjamin Bayer, SSN is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about *entities*, whereas other relations may represent facts about *relationships*. For example, a relation schema MAJORS (StudentSSN, DepartmentCode) asserts that students major in academic departments; a tuple in this relation relates a student to his or her major department. Hence, the relational model represents facts about both entities and relationships *uniformly* as relations.

An alternative interpretation of a relation schema is as a **predicate;** in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. This interpretation is quite useful in the context of logic programming languages, such as PROLOG, because it allows the relational model to be used within these languages. This is further discussed in Chapter 25 when we discuss deductive databases.

7.1.3 Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, \ldots, A_n)$.
- An n-tuple t in a relation r(R) is denoted by $t = <v_1, v_2, \ldots, v_n>$, where v_i is the value corresponding to attribute A_i. The following notation refers to **component values** of tuples:
  - Both $t[A_i]$ and $t.A_i$ refer to the value v_i in t for attribute A_i.
  - Both $t[A_u, A_w, \ldots, A_z]$ and $t.(A_u, A_w, \ldots, A_z)$, where $A_u, A_w, \ldots, A_z$ is a list of attributes from R, refer to the subtuple of values $<v_u, v_w, \ldots, v_z>$ from t corresponding to the attributes specified in the list.
- The letters Q, R, S denote relation names.
- The letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- In general, the name of a relation schema such as STUDENT *also indicates* the current set of tuples in that relation—the *current relation state*—whereas STUDENT(Name, SSN, . . .) refers *only* to the relation schema.

Petitioner Uber Ex-1052, 0222

- An attribute A can be qualified with the relation name R to which it belongs by using the *dot notation* R.A—for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.

As an example, consider the tuple t = <'Barbara Benson', '533-69-1238', '839-8461', '7384 Fontana Lane', null, 19, 3.25> from the STUDENT relation in Figure 7.1; we have t[Name] = <'Barbara Benson'>, and t[SSN, GPA, Age] = <'533-69-1238', 3.25, 19>.

7.2 Relational Constraints and Relational Database Schemas

In this section, we discuss the various restrictions on data that can be specified on a relational database schema in the form of constraints. These include domain constraints, key constraints, entity integrity, and referential integrity constraints. Other types of constraints, called *data dependencies* (which include *functional dependencies* and *multivalued dependencies*), are used mainly for database design by normalization and will be discussed in Chapters 14 and 15.

7.2.1 Domain Constraints

Domain constraints specify that the value of each attribute A must be an atomic value from the domain dom(A). We have already discussed the ways in which domains can be specified in Section 7.1.1. The data types associated with domains typically include standard numeric data types for integers (such as short-integer, integer, long-integer) and real numbers (float and double-precision float). Characters, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type where all possible values are explicitly listed. Rather than describe these in detail here, we discuss the data types offered by the SQL2 relational standard in Section 8.1.2.

7.2.2 Key Constraints and Constraints on Null

A *relation* is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples t_1 and t_2 in a relation state r of R, we have the constraint that

$$t_1[SK] \neq t_2[SK]$$

Any such set of attributes SK is called a **superkey** of the relation schema R. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in a state r of R can have

Petitioner Uber Ex-1052, 0223

the same value for SK. Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy. A **key** K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R. Hence, a key is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold.

For example, consider the STUDENT relation of Figure 7.1. The attribute set {SSN} is a key of STUDENT because no two student tuples can have the same value for SSN.[4] Any set of attributes that includes SSN—for example, {SSN, Name, Age}—is a superkey. However, the superkey {SSN, Name, Age} is not a key of STUDENT, because removing Name or Age or both from the set still leaves us with a superkey.

The value of a key attribute can be used to identify uniquely each tuple in the relation. For example, the SSN value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* relation state of the schema. A key is determined from the meaning of the attributes, and the property is *time-invariant*; it must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 7.1 as a key, because there is no guarantee that two students with identical names will never exist.[5]

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key.** For example, the CAR relation in Figure 7.4 has two candidate keys: LicenseNumber and EngineSerialNumber. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 7.4. Notice that, when a relation schema has several candidate keys, the choice of one to

CAR	LicenseNumber	EngineSerialNumber	Make	Model	Year
	Texas ABC-739	A69352	Ford	Mustang	96
	Florida TVP-347	B43696	Oldsmobile	Cutlass	99
	New York MPO-22	X83554	Oldsmobile	Delta	95
	California 432-TFY	C43742	Mercedes	190-D	93
	California RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

Figure 7.4 The CAR relation with two candidate keys: LicenseNumber and EngineSerialNumber.

4. Note that SSN is also a superkey.

5. Names are sometimes used as keys, but then some artifact—such as appending an ordinal number—must be used to distinguish between identical names.

Petitioner Uber Ex-1052, 0224

become primary key is arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes.

Another constraint on attributes specifies whether null values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-null value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

7.2.3 Relational Databases and Relational Database Schemas

So far, we have discussed single relations and single relation schemas. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema. A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \ldots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state**[6] DB of S is a set of relation states $DB = \{r_1, r_2, \ldots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC. Figure 7.5 shows a relational database schema

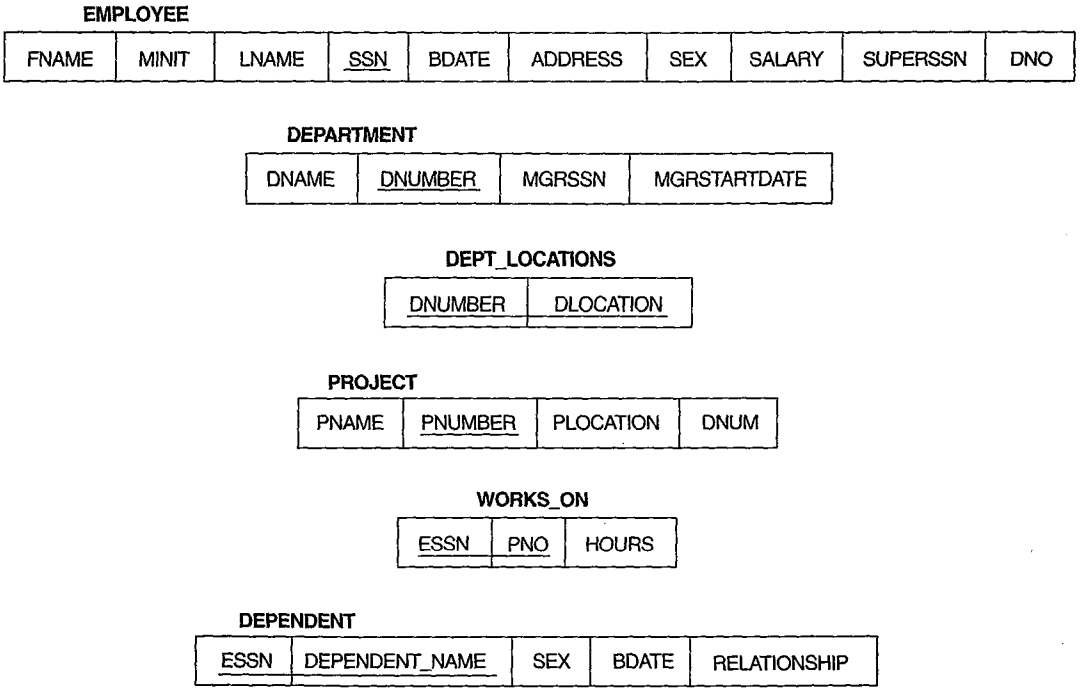


Figure 7.5 Schema diagram for the COMPANY relational database schema; the primary keys are underlined.

6. A relational database *state* is also called a relational database *instance*.

Petitioner Uber Ex-1052, 0225

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS

DNUMBER	DLOCATION
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

WORKS_ON

ESSN	PNO	HOURS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	null

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
333445555	Alice	F	1986-04-05	DAUGHTER
333445555	Theodore	M	1983-10-25	SON
333445555	Joy	F	1958-05-03	SPOUSE
987654321	Abner	M	1942-02-28	SPOUSE
123456789	Michael	M	1988-01-04	SON
123456789	Alice	F	1988-12-30	DAUGHTER
123456789	Elizabeth	F	1967-05-05	SPOUSE

Figure 7.6 One possible relational database state corresponding to the COMPANY schema.

that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. Figure 7.6 shows a relational database state corresponding to the COMPANY schema. We will use this schema and database state in this chapter and in Chapters 8 through 10 for developing example queries in different relational languages. When we refer to a relational database, we implicitly include both its schema and its current state.

Petitioner Uber Ex-1052, 0226

In Figure 7.5, the DNUMBER attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called DNO in EMPLOYEE and DNUM in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name NAME for both PNAME of PROJECT and DNAME of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different real-world concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of social security number appears twice in the EMPLOYEE relation of Figure 7.5: once in the role of the employee's social security number, and once in the role of the supervisor's social security number. We gave them distinct attribute names—SSN and SUPERSSN, respectively—in order to distinguish their meaning.

Each relational DBMS must have a Data Definition Language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. We present the SQL DDL in Section 8.1.

Integrity constraints are specified on a database schema and are expected to hold on every database state of that schema. In addition to domain and key constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

7.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be null. This is because the primary key value is used to identify individual tuples in a relation; having null values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had null for their primary keys, we might not be able to distinguish them.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples of the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 7.6, the attribute DNO of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the DNUMBER value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, we first define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation

Petitioner Uber Ex-1052, 0227

schema R_1 is a **foreign key** of R_1 that references relation R_2 if it satisfies the following two rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.
2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is null. In the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2. R_1 is called the **referencing relation** and R_2 is the **referenced relation.**

In a database of many relations, there are usually many referential integrity constraints. To specify these constraints, we must first have a clear understanding of the meaning or role that each set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 7.6. In the EMPLOYEE relation, the attribute DNO refers to the department for which an employee works; hence, we designate DNO to be a foreign key of EMPLOYEE, referring to the DEPARTMENT relation. This means that a value of DNO in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the DNUMBER attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of DNO *can be null* if the employee does not belong to a department. In Figure 7.6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department.

Notice that a foreign key can *refer to its own relation*. For example, the attribute SUPERSSN in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, SUPERSSN is a foreign key that references the EMPLOYEE relation itself. In Figure 7.6 the tuple for employee 'John Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith.'

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 7.7 shows the schema in Figure 7.5 with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema if we want to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key and entity integrity constraints, and make provisions to support referential integrity. These constraints are specified as a part of data definition.

The preceding integrity constraints do not include a large class of general constraints, sometimes called *semantic integrity constraints*, that may have to be specified and enforced on a relational database. Examples of such constraints are "the salary of an employee should not exceed the salary of the employee's supervisor" and "the maximum number of hours an employee can work on all projects per week is 56." Such constraints can be specified and enforced by using a general purpose **constraint specification language.** Mechanisms called

Petitioner Uber Ex-1052, 0228

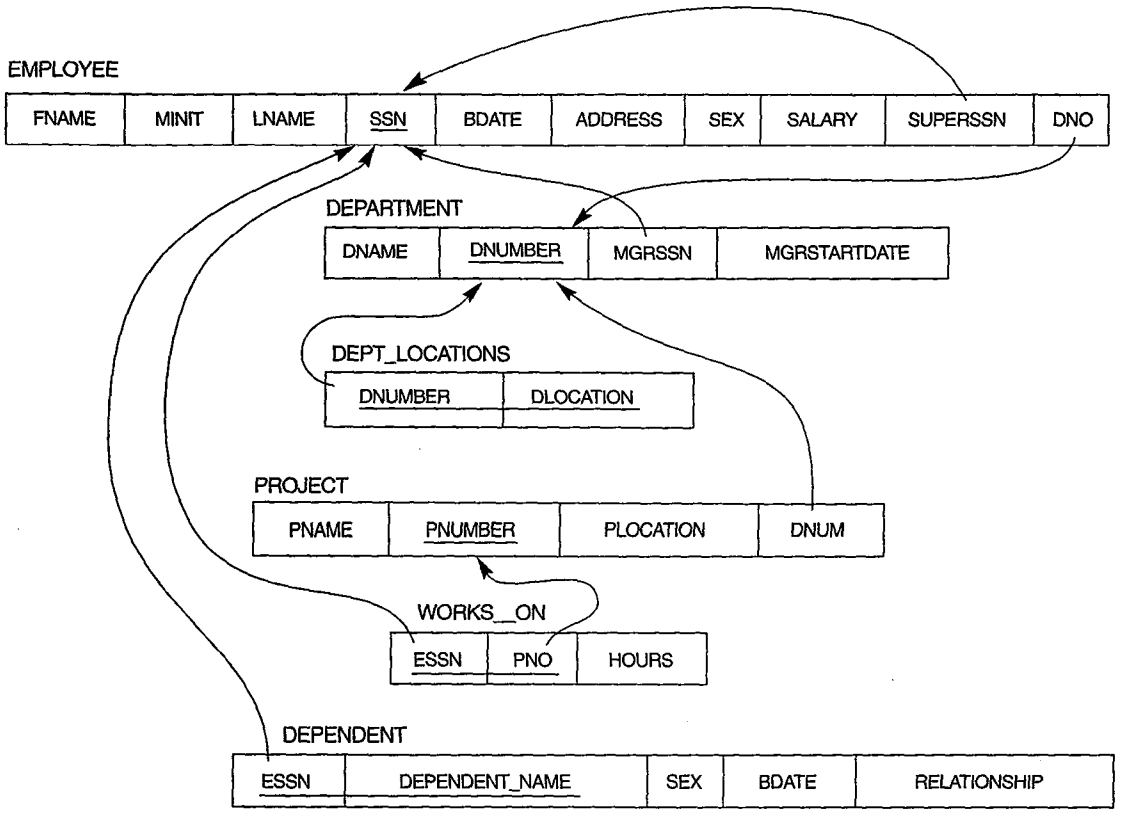


Figure 7.7 Referential integrity constraints displayed on the COMPANY relational database schema diagram.

triggers and **assertions** can be used. In SQL2, a CREATE ASSERTION statement is used for this purpose (see Chapters 8 and 23).

The types of constraints we discussed above may be termed as *state constraints*, because they define the constraints that a *valid state* of the database must satisfy. Another type of constraints, called *transition constraints*, can be defined to deal with state changes in the database.[7] An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically specified using active rules and triggers, as we shall discuss in Chapter 23.

7. State constraints are also called *static constraints*, and transition constraints are called *dynamic constraints*.

Petitioner Uber Ex-1052, 0229

7.3 Update Operations and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates*. The relational algebra operations, which can be used to specify retrievals, are discussed in detail in Section 7.4. In this section, we concentrate on the update operations. There are three basic update operations on relations: (1) insert, (2) delete, and (3) modify. **Insert** is used to insert a new tuple or tuples in a relation; **Delete** is used to delete tuples; and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples. Whenever update operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each update operation and the types of actions that may be taken if an update does cause a violation. We use the database shown in Figure 7.6 for examples and discuss only key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 7.7. For each type of update, we give some example operations and discuss any constraints that each operation may violate.

7.3.1 The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R. Insert can violate any of the four types of constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R). Entity integrity can be violated if the primary key of the new tuple t is null. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

1. Insert <'Cecilia', 'F', 'Kolonsky', null, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, null, 4> into EMPLOYEE.
   - This insertion violates the entity integrity constraint (null for the primary key SSN), so it is rejected.
2. Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.
   - This insertion violates the key constraint because another tuple with the same SSN value already exists in the EMPLOYEE relation, and so it is rejected.
3. Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
   - This insertion violates the referential integrity constraint specified on DNO because no DEPARTMENT tuple exists with DNUMBER = 7.

Petitioner Uber Ex-1052, 0230

4. Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, null, 4> into EMPLOYEE.
   - This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to *reject the insertion*. In this case, it would be useful if the DBMS could explain to the user why the insertion was rejected. Another option is to attempt to *correct the reason for rejecting the insertion*, but this is typically not used for violations caused by Insert; rather, it is used more often in correcting violations for Delete and Update. The following examples illustrate how this option may be used for Insert violations. In operation 1 above, the DBMS could ask the user to provide a value for SSN and could accept the insertion if a valid SSN value were provided. In operation 3, the DBMS could either ask the user to change the value of DNO to some valid value (or set it to null), or it could ask the user to insert a DEPARTMENT tuple with DNUMBER = 7 and could accept the insertion only after such an operation was accepted. Notice that in the latter case the insertion can **cascade** back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for MGRSSN that does not exist in the EMPLOYEE relation.

7.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity, if the tuple being deleted is referenced by the foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

1. Delete the WORKS_ON tuple with ESSN = '999887777' and PNO = 10.
   - This deletion is acceptable.
2. Delete the EMPLOYEE tuple with SSN = '999887777'.
   - This deletion is not acceptable, because tuples in WORKS_ON refer to this tuple. Hence, if the tuple is deleted, referential integrity violations will result.
3. Delete the EMPLOYEE tuple with SSN = '333445555'.
   - This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Three options are available if a deletion operation causes a violation. The first option is to *reject the deletion*. The second option is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with ESSN = '999887777'. A third option is to *modify the referencing attribute values* that cause the violation; each such value is either set to null or changed to reference another valid tuple. Notice that, if a referencing attribute that causes a violation is *part of the primary key*, it *cannot be* set to null; otherwise, it would violate entity integrity.

Petitioner Uber Ex-1052, 0231

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with ESSN = '333445555'. Tuples in EMPLOYEE with SUPERSSN = '333445555' and the tuple in DEPARTMENT with MGRSSN = '333445555' can have their SUPERSSN and MGRSSN values changed to other valid values or to null. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple. In general, when a referential integrity constraint is specified, the DBMS should allow the user to *specify which of the three options* applies in case of a violation of the constraint. We discuss how to specify these options in SQL2 DDL in Chapter 8.

7.3.3 The Update Operation

The **Update** operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation R. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

1. Update the SALARY of the EMPLOYEE tuple with SSN = '999887777' to 28000.
   - Acceptable.
2. Update the DNO of the EMPLOYEE tuple with SSN = '999887777' to 1.
   - Acceptable.
3. Update the DNO of the EMPLOYEE tuple with SSN = '999887777' to 7.
   - Unacceptable, because it violates referential integrity.
4. Update the SSN of the EMPLOYEE tuple with SSN = '999887777' to '987654321'.
   - Unacceptable, because it violates primary key and referential integrity constraints.

Updating an attribute that is neither a primary key nor a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place, because we use the primary key to identify tuples. Hence, the issues discussed earlier under both Insert and Delete come into play. If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is null).

7.4 Basic Relational Algebra Operations

In addition to defining the database structure and constraints, a data model must include a set of operations to manipulate the data. A basic set of relational model operations constitute the **relational algebra.** These operations enable the user to specify basic retrieval requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be

Petitioner Uber Ex-1052, 0232

further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression,** whose result will also be a relation.

The relational algebra operations are usually divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases; these include SELECT, PROJECT, and JOIN, among others. The SELECT and PROJECT operations are discussed first, because they are the simplest. Then we discuss set operations. Finally, we discuss JOIN and other complex operations. The relational database shown in Figure 7.6 is used for our examples.

Some common database requests cannot be performed with the basic relational algebra operations, so additional operations are needed to express these requests. Some of these additional operations are described in Section 7.5.

7.4.1 The SELECT Operation

The **SELECT** operation is used to select a *subset* of the tuples from a relation that satisfy a **selection condition.** One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$\sigma_{DNO=4}(EMPLOYEE)$

$\sigma_{SALARY>30000}(EMPLOYEE)$

In general, the SELECT operation is denoted by

$\sigma_{<\text{selection condition}>}(R)$

where the symbol σ (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R. Notice that R is generally a *relational algebra expression* whose result is a relation; the simplest expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as R. The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>, or

<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators $\{=, <, \leq, >, \geq, \neq\}$, and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the Boolean operators AND, OR, and NOT to form a general selection condition. For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

$\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(EMPLOYEE)$

Petitioner Uber Ex-1052, 0233

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 FireOak,Humble,TX	M	38000	333445555	5

(b)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Figure 7.8 Results of SELECT and PROJECT operations. (a) $\sigma_{(DNO=4\ AND\ SALARY>25000)\ OR\ (DNO=5\ AND\ SALARY>30000)}(EMPLOYEE)$. (b) $\pi_{LNAME,\ FNAME,\ SALARY}(EMPLOYEE)$. (c) $\pi_{SEX,\ SALARY}(EMPLOYEE)$

The result is shown in Figure 7.8(a). Notice that the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$ apply to attributes whose domains are *ordered values*, such as numeric or date domains. Domains of strings of characters are considered ordered based on the collating sequence of the characters. If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set $\{=, \neq\}$ can be used. An example of an unordered domain is the domain Color = {red, blue, green, white, yellow, . . .} where no order is specified among the various colors. Some domains allow additional types of comparison operators; for example, a domain of character strings may allow the comparison operator SUBSTRING_OF.

In general, the result of a SELECT operation can be determined as follows. The <selection condition> is applied independently to each tuple t in R. This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to true, then tuple t is **selected.** All the selected tuples appear in the result of the SELECT operation. The Boolean conditions AND, OR, and NOT have their normal interpretation as follows:

- (cond1 AND cond2) is true if both (cond1) and (cond2) are true; otherwise, it is false.
- (cond1 OR cond2) is true if either (cond1) or (cond2) or both are true; otherwise, it is false.
- (NOT cond) is true if cond is false; otherwise, it is false.

The SELECT operator is unary; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually;* hence, selection conditions cannot involve more than one tuple. The **degree** of the relation resulting from a SELECT operation is the same as that of R. The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in R. That is, $|\sigma_c(R)| \leq |R|$ for any condition C.

Petitioner Uber Ex-1052, 0234

The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative;** that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is:

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ldots)) = \sigma_{<cond1> \text{ AND } <cond2> \text{ AND } \ldots \text{ AND } <condn>}(R)$$

7.4.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation selects some of the *rows* from the table while discarding other rows. The PROJECT operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$$

The resulting relation is shown in Figure 7.8(b). The general form of the PROJECT operation is

$$\pi_{<\text{attribute list}>}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation and <attribute list> is a list of attributes from the attributes of relation R. Again, notice that R is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in <attribute list> and *in the same order as they appear in the list.* Hence, its **degree** is equal to the number of attributes in <attribute list>.

If the attribute list includes only nonkey attributes of R, duplicate tuples are likely to occur; the PROJECT operation *removes any duplicate tuples,* so the result of the PROJECT operation is a set of tuples and hence a valid relation.[8] This is known as **duplicate elimination.** For example, consider the following PROJECT operation:

$$\pi_{\text{SEX, SALARY}}(\text{EMPLOYEE})$$

The result is shown in Figure 7.8(c). Notice that the tuple <F, 25000> appears only once in Figure 7.8(c), even though this combination of values appears twice in the EMPLOYEE relation.

8. If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. As we shall see in Chapter 8, the SQL language allows the user to specify whether duplicates should be eliminated or not.

Petitioner Uber Ex-1052, 0235

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R. If the projection list is a superkey of R—that is, it includes some key of R—the resulting relation has the *same number* of tuples as R. Moreover,

$$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$$

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

7.4.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 7.8 do not have any names. In general, we may want to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must name the relations that hold the intermediate results. For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression as follows:

$$\pi_{FNAME, LNAME, SALARY}(\sigma_{DNO=5}(EMPLOYEE))$$

Figure 7.9(a) shows the result of this relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation:

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$

$$RESULT \leftarrow \pi_{FNAME, LNAME, SALARY}(DEP5\_EMPS)$$

It is often simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to **rename** the attributes in the intermediate and result relations. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$TEMP \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$

$$R(FIRSTNAME, LASTNAME, SALARY) \leftarrow \pi_{FNAME, LNAME, SALARY}(TEMP)$$

The above two operations are illustrated in Figure 7.9(b). If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

We can also define a RENAME operation—which can rename either the relation name, or the attribute names, or both—in a manner similar to the way we defined SELECT

Petitioner Uber Ex-1052, 0236

(a)

FNAME	LNAME	SALARY
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren,Houston,TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak,Humble,TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice,Houston,TX	F	25000	333445555	5

R	FIRSTNAME	LASTNAME	SALARY
	John	Smith	30000
	Franklin	Wong	40000
	Ramesh	Narayan	38000
	Joyce	English	25000

Figure 7.9 Results of relational algebra expressions.
(a) $\pi_{\text{LNAME, FNAME, SALARY}}(\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$.
(b) The same expression using intermediate relations and renaming of attributes.

and PROJECT. The general RENAME operation when applied to a relation R of degree n is denoted by

$$\rho_{S(B1, B2, \ldots, Bn)}(R) \text{ or } \rho_S(R) \text{ or } \rho_{(B1, B2, \ldots, Bn)}(R)$$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and $B_1, B_2, \ldots, B_n$ are the new attribute names. The first expression renames both the relation and its attributes; the second renames the relation only; and the third renames the attributes only. If the attributes of R are $(A_1, A_2, \ldots, A_n)$ in that order, then each A_i is renamed as B_i.

7.4.4 Set Theoretic Operations

The next group of relational algebra operations are the standard mathematical operations on sets. For example, to retrieve the social security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$$

$$\text{RESULT1} \leftarrow \pi_{\text{SSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT2(SSN)} \leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$$

The relation RESULT1 has the social security numbers of all employees who work in department 5, whereas RESULT2 has the social security numbers of all employees who

Petitioner Uber Ex-1052, 0237

RESULT1	SSN
	123456789
	333445555
	666884444
	453453453

RESULT2	SSN
	333445555
	888665555

RESULT	SSN
	123456789
	333445555
	666884444
	453453453
	888665555

Figure 7.10 Query result after the UNION operation: RESULT ← RESULT1 ∪ RESULT2.

directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 7.10).

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION, INTERSECTION,** and **SET DIFFERENCE.** These are binary operations; that is, each is applied to two sets. When these operations are adapted to relational databases, the two relations on which any of the above three operations are applied must have the same **type of tuples;** this condition is called *union compatibility*. Two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_n)$ are said to be **union compatible** if they have the same degree n, and if $dom(A_i) = dom(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and that each pair of corresponding attributes have the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

- UNION: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- INTERSECTION: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
- SET DIFFERENCE: The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

We will adopt the convention that the resulting relation has the same attribute names as the *first* relation R. Figure 7.11 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 7.11(a) are union compatible, and their tuples represent the names of students and instructors, respectively. The result of the UNION operation in Figure 7.11(b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 7.11c) includes only those who are both students and instructors. Notice that both UNION and INTERSECTION are *commutative operations;* that is

$$R \cup S = S \cup R, \text{ and } R \cap S = S \cap R$$

Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are *associative operations;* that is

$$R \cup (S \cup T) = (R \cup S) \cup T, \text{ and } (R \cap S) \cap T = R \cap (S \cap T)$$

Petitioner Uber Ex-1052, 0238

(a)

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	FNAME	LNAME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

(b)

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

FN	LN
Susan	Yao
Ramesh	Shah

(d)

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

Figure 7.11 Illustrating the set operations UNION, INTERSECTION, and DIFFERENCE. (a) Two union compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT − INSTRUCTOR. (e) INSTRUCTOR − STUDENT.

The DIFFERENCE operation is *not commutative*; that is, in general

$$R - S \neq S - R$$

Figure 7.11(d) shows the names of students who are not instructors, and Figure 7.11(e) shows the names of instructors who are not students.

Next we discuss the **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—denoted by **x**, which is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. This operation is used to combine tuples from two relations in a combinatorial fashion. In general, the result of $R(A_1,$

Petitioner Uber Ex-1052, 0239

$A_2, \ldots, A_n$) x S($B_1, B_2, \ldots, B_m$) is a relation Q with n + m attributes Q($A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$), in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples and S has n_S tuples, then R x S will have $n_R * n_S$ tuples. The operation applied by itself is generally meaningless. It is useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve for each female employee a list of the names of her dependents; we can do this as follows:

FEMALE_EMPS $\leftarrow \sigma_{SEX='F'}$(EMPLOYEE)

EMPNAMES $\leftarrow \pi_{FNAME, LNAME, SSN}$(FEMALE_EMPS)

EMP_DEPENDENTS $\leftarrow$ EMPNAMES x DEPENDENT

ACTUAL_DEPENDENTS $\leftarrow \sigma_{SSN=ESSN}$(EMP_DEPENDENTS)

RESULT $\leftarrow \pi_{FNAME, LNAME, DEPENDENT\_NAME}$(ACTUAL_DEPENDENTS)

The resulting relations from the above sequence of operations are shown in Figure 7.12. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 7.12 with DEPENDENT from Figure 7.6. In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful. We only want to combine a female employee tuple with her dependents—namely, the DEPENDENT tuples whose ESSN values match the SSN value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can then SELECT only related tuples from the two relations by specifying an appropriate selection condition, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is used quite commonly to identify and select related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation. We discuss the JOIN operation next.

7.4.5 The JOIN Operation

The JOIN operation, denoted by $\bowtie$, is used to combine *related tuples* from two relations into single tuples. This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations. To illustrate join, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose SSN value matches the MGRSSN value in the department tuple. We do this by using the JOIN operation, and then projecting the result over the necessary attributes, as follows:

DEPT_MGR $\leftarrow$ DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

RESULT $\leftarrow \pi_{DNAME, LNAME, FNAME}$(DEPT_MGR)

The first operation is illustrated in Figure 7.13. Note that MGRSSN is a foreign key and that the referential integrity constraint plays a role in having matching tuples in the

Petitioner Uber Ex-1052, 0240

FEMALE_EMPS	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle,Spring,TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
	Joyce	A	English	453453453	1972-07-31	5631 Rice,Houston,TX	F	25000	333445555	5

EMPNAMES	FNAME	LNAME	SSN
	Alicia	Zelaya	999887777
	Jennifer	Wallace	987654321
	Joyce	English	453453453

EMP_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	• • •
	Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	• • •
	Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	• • •
	Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	• • •
	Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	• • •
	Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	• • •
	Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	• • •
	Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	• • •
	Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	• • •
	Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	• • •
	Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	• • •
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	• • •
	Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	• • •
	Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	• • •
	Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	• • •
	Joyce	English	453453453	333445555	Alice	F	1986-04-05	• • •
	Joyce	English	453453453	333445555	Theodore	M	1983-10-25	• • •
	Joyce	English	453453453	333445555	Joy	F	1958-05-03	• • •
	Joyce	English	453453453	987654321	Abner	M	1942-02-28	• • •
	Joyce	English	453453453	123456789	Michael	M	1988-01-04	• • •
	Joyce	English	453453453	123456789	Alice	F	1988-12-30	• • •
	Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	• • •

ACTUAL_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	• • •
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	• • •

RESULT	FNAME	LNAME	DEPENDENT_NAME
	Jennifer	Wallace	Abner

Figure 7.12 An illustration of the CARTESIAN PRODUCT operation.

DEPT_MGR	DNAME	DNUMBER	MGRSSN	• • •	FNAME	MINIT	LNAME	SSN	• • •
	Research	5	333445555	• • •	Franklin	T	Wong	333445555	• • •
	Administration	4	987654321	• • •	Jennifer	S	Wallace	987654321	• • •
	Headquarters	1	888665555	• • •	James	E	Borg	888665555	• • •

Figure 7.13 Illustrating the JOIN operation.

Petitioner Uber Ex-1052, 0241

referenced relation EMPLOYEE. The example we gave earlier to illustrate the CARTESIAN PRODUCT operation can be specified, using the JOIN operation, by replacing the two operations:

EMP_DEPENDENTS ← EMPNAMES X DEPENDENT

ACTUAL_DEPENDENTS ← $\sigma_{SSN=ESSN}$(EMP_DEPENDENTS)

with a single JOIN operation:

ACTUAL_DEPENDENTS ← EMPNAMES $\bowtie_{SSN=ESSN}$ DEPENDENT

The general form of a JOIN operation on two relations[9] $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is:

$R \bowtie_{\text{<join condition>}} S$

The result of the JOIN is a relation Q with n + m attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition*. This is the main difference between CARTESIAN PRODUCT and JOIN: in JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to true is included in the resulting relation Q *as a single combined tuple*.

A general join condition is of the form:

<condition> AND <condition> AND . . . AND <condition>

where each condition is of the form $A_i\ \theta\ B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}. A JOIN operation with such a general join condition is called a **THETA JOIN.** Tuples whose join attributes are null *do not* appear in the result. In that sense, the join operation does *not* necessarily preserve all of the information in the participating relations.

The most common JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN.** Both examples we have considered were EQUIJOINs. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple. For example, in Figure 7.13, the values of the attributes MGRSSN and SSN are identical in every tuple of DEPT_MGR because of the equality join condition specified on these two attributes. Because one of each pair of attributes with identical values is superfluous, a new operation called NATURAL JOIN—denoted by * —was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.[10] The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have

9. Again, notice that R and S can be the relations that result from general *relational algebra expressions*.

10. NATURAL JOIN is basically an EQUIJOIN followed by removal of the superfluous attributes.

Petitioner Uber Ex-1052, 0242

(a)

PROJ_DEPT	PNAME	PNUMBER	PLOCATION	DNUM	DNAME	MGRSSN	MGRSTARTDATE
	ProductX	1	Bellaire	5	Research	333445555	1988-05-22
	ProductY	2	Sugarland	5	Research	333445555	1988-05-22
	ProductZ	3	Houston	5	Research	333445555	1988-05-22
	Computerization	10	Stafford	4	Administration	987654321	1995-01-01
	Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
	Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	LOCATION
	Headquarters	1	888665555	1981-06-19	Houston
	Administration	4	987654321	1995-01-01	Stafford
	Research	5	333445555	1988-05-22	Bellaire
	Research	5	333445555	1988-05-22	Sugarland
	Research	5	333445555	1988-05-22	Houston

Figure 7.14 An illustration of the NATURAL JOIN operation. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

the same name in both relations. If this is not the case, a renaming operation is applied first. In the following example, we first rename the DNUMBER attribute of DEPARTMENT to DNUM—so that it has the same name as the DNUM attribute in PROJECT—then apply NATURAL JOIN:

$$\text{PROJ\_DEPT} \leftarrow \text{PROJECT} * \rho_{(\text{DNAME,DNUM,MGRSSN,MGRSTARTDATE})}(\text{DEPARTMENT})$$

The attribute DNUM is called the **join attribute.** The resulting relation is illustrated in Figure 7.14(a). In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but *only one join attribute* is kept.

If the attributes on which the natural join is specified *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the DNUMBER attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write:

$$\text{DEPT\_LOCS} \leftarrow \text{DEPARTMENT} * \text{DEPT\_LOCATIONS}$$

The resulting relation is shown in Figure 7.14(b), which combines each department with its locations and has one tuple for each location. In general, NATURAL JOIN is performed by equating *all* attribute pairs that have the same name in the two relations. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

A more general but non-standard definition for NATURAL JOIN is

$$Q \leftarrow R *_{(\langle list1 \rangle),(\langle list2 \rangle)} S$$

Petitioner Uber Ex-1052, 0243

In this case, <list1> specifies a list of i attributes from R, and <list2> specifies a list of i attributes from S. The lists are used to form equality comparison conditions between pairs of corresponding attributes; the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R—<list 1>—is kept in the result Q.

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie_{\text{<join condition>}} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity,** which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN becomes a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

The join operation is used to combine data from multiple relations so that related information can be presented in a single table. Note that sometimes a join may be specified between a relation and itself, as we shall illustrate in Section 7.5.2. The natural join or equijoin operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

$$((\text{PROJECT} \bowtie_{\text{DNUM=DNUMBER}} \text{DEPARTMENT}) \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE})$$

This links each project to its controlling department, and then relates the department to its manager employee. The net result is a consolidated relation where each tuple contains this project-department-manager information.

7.4.6 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, -, \times\}$ is a **complete** set; that is, any of the other relational algebra operations can be expressed as a *sequence of operations from this set*. For example, the INTERSECTION operation can be expressed by using UNION and DIFFERENCE as follows:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \bowtie_{\text{<condition>}} S \equiv \sigma_{\text{<condition>}} (R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra; however, they are very important because they are convenient to use and are very commonly applied in database applications. Other operations have been included in the relational

Petitioner Uber Ex-1052, 0244

algebra for convenience rather than necessity. We discuss one of these—the DIVISION operation—in the next section.

7.4.7 The DIVISION Operation

The DIVISION operation is useful for a special kind of query that sometimes occurs in database applications. An example is "Retrieve the names of employees who work on *all* the projects that 'John Smith' works on." To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$\text{SMITH} \leftarrow \sigma_{\text{FNAME='John' AND LNAME='Smith'}}(\text{EMPLOYEE})$$
$$\text{SMITH\_PNOS} \leftarrow \pi_{\text{PNO}}(\text{WORKS\_ON} \bowtie_{\text{ESSN=SSN}} \text{SMITH})$$

Next, create a relation that includes a tuple <PNO, ESSN> whenever the employee whose social security number is ESSN works on the project whose number is PNO in the intermediate relation SSN_PNOS:

$$\text{SSN\_PNOS} \leftarrow \pi_{\text{ESSN,PNO}}(\text{WORKS\_ON})$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' social security numbers:

$$\text{SSNS(SSN)} \leftarrow \text{SSN\_PNOS} \div \text{SMITH\_PNOS}$$
$$\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME}}(\text{SSNS} * \text{EMPLOYEE})$$

The previous operations are shown in Figure 7.15(a). In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where $X \subseteq Z$. Let $Y = Z - X$ (and hence $Z = X \cup Y$); that is, let Y be the set of attributes of R that are not attributes of S. The result of DIVISION is a relation T(Y) that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ *for every tuple* t_S in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every* tuple in S.

Figure 7.15(b) illustrates a DIVISION operator where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$. Notice that the tuples (values) b_1 and b_4 appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: b_2 does not appear with a_2 and b_3 does not appear with a_1.

The DIVISION operator can be expressed as a sequence of π, $\times$, and $-$ operations as follows:

$$T_1 \leftarrow \pi_Y(R)$$
$$T_2 \leftarrow \pi_Y((S \times T_1) - R)$$
$$T \leftarrow T_1 - T_2$$

Petitioner Uber Ex-1052, 0245

(a)

SSN_PNOS	ESSN	PNO
	123456789	1
	123456789	2
	666884444	3
	453453453	1
	453453453	2
	333445555	2
	333445555	3
	333445555	10
	333445555	20
	999887777	30
	999887777	10
	987987987	10
	987987987	30
	987654321	30
	987654321	20
	888665555	20

SMITH_PNOS	PNO
	1
	2

SSNS	SSN
	123456789
	453453453

(b)

R	A	B
	a1	b1
	a2	b1
	a3	b1
	a4	b1
	a1	b2
	a3	b2
	a2	b3
	a3	b3
	a4	b3
	a1	b4
	a2	b4
	a3	b4

S	A
	a1
	a2
	a3

T	B
	b1
	b4

Figure 7.15 Illustrating the DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

Petitioner Uber Ex-1052, 0246

7.5 Additional Relational Operations

Some common database requests—which are needed in commercial query languages for relational DBMSs—cannot be performed with the basic relational algebra operations described in Section 7.4. In this section we define additional operations to express these requests. These operations enhance the expressive power of the relational algebra.

7.5.1 Aggregate Functions and Grouping

The first type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the number of employee tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. An example would be to group employee tuples by DNO, so that each group includes the tuples for employees working in the same department. We can then list each DNO value along with, say, the average salary of employees within the department.

We can define an AGGREGATE FUNCTION operation, using the symbol $\mathfrak{F}$ (pronounced "script F"),[11] to specify these types of requests as follows:

$$\text{<grouping attributes>}\ \mathfrak{F}\ \text{<function list>}\ (R)$$

where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and <attribute> is an attribute of the relation specified by R. The resulting relation has the grouping attributes plus one attribute for each element in the function list. For example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

$$\rho_{R(\text{DNO, NO\_OF\_EMPLOYEES, AVERAGE\_SAL})}\ (\text{DNO}\ \mathfrak{F}\ \text{COUNT SSN, AVERAGE SALARY}\ (\text{EMPLOYEE}))$$

The result of this operation is shown in Figure 7.16(a).

In the above example, we specified a list of attribute names—between parentheses in the rename operation—for the resulting relation R. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>. For example, Figure 7.16(b) shows the result of the following operation:

$$\text{DNO}\ \mathfrak{F}\ \text{COUNT SSN, AVERAGE SALARY}(\text{EMPLOYEE})$$

11. There is no single agreed-upon notation for specifying aggregate functions. In some cases a "script A" is used.

Petitioner Uber Ex-1052, 0247

(a)

R	DNO	NO_OF_EMPLOYEES	AVERAGE_SAL
	5	4	33250
	4	3	31000
	1	1	55000

(b)

DNO	COUNT_SSN	AVERAGE_SALARY
5	4	33250
4	3	31000
1	1	55000

(c)

COUNT_SSN	AVERAGE_SALARY
8	35125

Figure 7.16 An illustration of the AGGREGATE FUNCTION operation. (a) $R(\text{DNO, NO\_OF\_EMPLOYEES, AVERAGE\_SAL}) \leftarrow {}_{\text{DNO}}\Im_{\text{COUNT SSN, AVERAGE SALARY}}(\text{EMPLOYEE})$. (b) ${}_{\text{DNO}}\Im_{\text{COUNT SSN, AVERAGE SALARY}}(\text{EMPLOYEE})$. (c) $\Im_{\text{COUNT SSN, AVERAGE SALARY}}(\text{EMPLOYEE})$.

If no grouping attributes are specified, the functions are applied to the attribute values of *all the tuples* in the relation, so the resulting relation has a *single tuple only*. For example, Figure 7.16(c) shows the result of the following operation:

$$\Im_{\text{COUNT SSN, AVERAGE SALARY}}(\text{EMPLOYEE})$$

It is important to note that, in general, duplicates are *not eliminated* when an aggregate function is applied; this way, the normal interpretation of functions such as SUM and AVERAGE is computed.[12] It is worth emphasizing that the result of applying an aggregate function is a relation, not a scalar number—even if it has a single value.

7.5.2 Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic relational algebra is **recursive closure.** This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor. This relationship is described by the foreign key SUPERSSN of the EMPLOYEE relation in Figures 7.6 and 7.7, which relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of an employee e at all levels—that is, all employees e′ directly supervised by e; all employees e″ directly supervised by each employee e′; all employees e‴ directly supervised by each employee e″; and so on. Although it is

12. In SQL, the option of eliminating duplicates before applying the aggregate function is available by including the keyword DISTINCT (see Chapter 8).

Petitioner Uber Ex-1052, 0248

straightforward in the relational algebra to specify all employees supervised by e *at a specific level*, it is difficult to specify all supervisees at *all* levels. For example, to specify the SSNs of all employees e′ directly supervised—*at level one*—by the employee e whose name is 'James Borg' (see Figure 7.6), we can apply the following operation:

$$\text{BORG\_SSN} \leftarrow \pi_{\text{SSN}}(\sigma_{\text{FNAME='James' AND LNAME='Borg'}}(\text{EMPLOYEE}))$$

$$\text{SUPERVISION}(\text{SSN1, SSN2}) \leftarrow \pi_{\text{SSN, SUPERSSN}}(\text{EMPLOYEE})$$

$$\text{RESULT1}(\text{SSN}) \leftarrow \pi_{\text{SSN1}}(\text{SUPERVISION} \bowtie_{\text{SSN2=SSN}} \text{BORG\_SSN})$$

To retrieve all employees supervised by Borg at level 2—that is, all employees e″ supervised by some employee e′ who is directly supervised by Borg—we can apply another JOIN to the result of the first query, as follows:

$$\text{RESULT2}(\text{SSN}) \leftarrow \pi_{\text{SSN1}}(\text{SUPERVISION} \bowtie_{\text{SSN2=SSN}} \text{RESULT1})$$

To get both sets of employees supervised at levels 1 and 2 by 'James Borg,' we can apply the UNION operation to the two results, as follows:

$$\text{RESULT} \leftarrow \text{RESULT2} \cup \text{RESULT1}$$

The results of these queries are illustrated in Figure 7.17. Although it is possible to retrieve employees at each level and then take their UNION, we cannot, in general, specify a query such as "retrieve the supervisees of 'James Borg' at all levels" without utilizing a looping mechanism.[13] An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

7.5.3 OUTER JOIN and OUTER UNION Operations

Finally, we discuss some extensions of the JOIN and UNION operations. The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation R * S, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result. Tuples with null in the join attributes are also eliminated. A set of operations, called OUTER JOINs, can be used when we want to keep all the tuples in R, or those in S, or those in both relations in the result of the JOIN, whether or not they have matching tuples in the other relation. This satisfies the need of queries where tuples from two tables are to be combined by matching corresponding rows, but some tuples are liable to be lost for lack of matching values. In such cases an operation is desirable that would preserve all the tuples whether or not they produce a match.

For example, suppose that we want a list of all employee names and also the name of the departments they manage *if they happen to manage a department*; we can apply an operation **LEFT OUTER JOIN**, denoted by ⟕, to retrieve the result as follows:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} \text{ ⟕}_{\text{SSN=MGRSSN}} \text{DEPARTMENT})$$

$$\text{RESULT} \leftarrow \pi_{\text{FNAME, MINIT, LNAME, DNAME}}(\text{TEMP})$$

13. We will discuss recursive queries further in Chapter 25 when we give an overview of deductive databases. Also, the SQL3 standard includes syntax for recursive closure.

Petitioner Uber Ex-1052, 0249

(Borg's SSN is 888665555)

	(SSN)	(SUPERSSN)
SUPERVISION	SSN1	SSN2
	123456789	333445555
	333445555	888665555
	999887777	987654321
	987654321	888665555
	666884444	333445555
	453453453	333445555
	987987987	987654321
	888665555	null

RESULT 1	SSN
	333445555
	987654321

(Supervised by Borg)

RESULT 2	SSN
	123456789
	999887777
	666884444
	453453453
	987987987

(Supervised by Borg's subordinates)

RESULT	SSN
	123456789
	999887777
	666884444
	453453453
	987987987
	333445555
	987654321

(RESULT1 ∪ RESULT2)

Figure 7.17 A two-level recursive query.

The LEFT OUTER JOIN operation keeps every tuple in the *first* or *left* relation R in R ⟕ S; if no matching tuple is found in S, then the attributes of S in the join result are filled or "padded" with null values. The result of these operations is shown in Figure 7.18.

A similar operation, RIGHT OUTER JOIN, denoted by ⟖, keeps every tuple in the *second* or right relation S in the result of R ⟖ S. A third operation, FULL OUTER JOIN, denoted by ⟗, keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with null values as needed. The three outer join operations are part of the SQL2 standard (see Chapter 8).

The OUTER UNION operation was developed to take the union of tuples from two relations if the relations are *not union compatible*. This operation will take the UNION of tuples in two relations that are **partially compatible,** meaning that only some of their attributes are union compatible. It is expected that the list of compatible attributes includes a key for both relations. Tuples from the component relations with the same key are represented only once in the result and have values for all attributes in the result. The attributes that are not union compatible from either relation are kept in the result, and tuples that have no values for these attributes are padded with null values. For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, SSN, Department, Advisor) and FACULTY(Name, SSN, Department, Rank). The resulting relation schema is R(Name, SSN, Department, Advisor, Rank), and all the tuples from both

Petitioner Uber Ex-1052, 0250

RESULT	FNAME	MINIT	LNAME	DNAME
	John	B	Smith	null
	Franklin	T	Wong	Research
	Alicia	J	Zelaya	null
	Jennifer	S	Wallace	Administration
	Ramesh	K	Narayan	null
	Joyce	A	English	null
	Ahmad	V	Jabbar	null
	James	E	Borg	Headquarters

Figure 7.18 The LEFT OUTER JOIN operation.

relations are included in the result. Student tuples will have a null for the Rank attribute, whereas faculty tuples will have a null for the Advisor attribute. A tuple that exists in both will have values for all its attributes.[14]

Another capability that exists in most commercial languages (but not in the basic relational algebra) is that of specifying operations on values after they are extracted from the database. For example, arithmetic operations such as +, −, and * can be applied to numeric values.

7.6 Examples of Queries in Relational Algebra

We now give additional examples to illustrate the use of the relational algebra operations. All examples refer to the database of Figure 7.6. In general, the same query can be stated in numerous ways using the various operations. We will state each query in one way and leave it to the reader to come up with equivalent formulations.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

$$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{DNAME='Research'}}(\text{DEPARTMENT})$$

$$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{DNUMBER=DNO}} \text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME, ADDRESS}}(\text{RESEARCH\_EMPS})$$

This query could be specified in other ways; for example, the order of the JOIN and SELECT operations could be reversed, or the JOIN could be replaced by a NATURAL JOIN (after renaming).

14. Notice that OUTER UNION is equivalent to a FULL OUTER JOIN if the join attributes are *all* the common attributes of the two relations.

Petitioner Uber Ex-1052, 0251

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{PLOCATION='Stafford'}}(\text{PROJECT})$$

$$\text{CONTR\_DEPT} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{DNUM=DNUMBER}} \text{DEPARTMENT})$$

$$\text{PROJ\_DEPT\_MGR} \leftarrow (\text{CONTR\_DEPT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}(\text{PROJ\_DEPT\_MGR})$$

QUERY 3

Find the names of employees who work on *all* the projects controlled by department number 5.

$$\text{DEPT5\_PROJS}(\text{PNO}) \leftarrow \pi_{\text{PNUMBER}}(\sigma_{\text{DNUM=5}}(\text{PROJECT}))$$

$$\text{EMP\_PRJO}(\text{SSN, PNO}) \leftarrow \pi_{\text{ESSN, PNO}}(\text{WORKS\_ON})$$

$$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PRJO} \div \text{DEPT5\_PROJS}$$

$$\text{RESULT} \leftarrow \pi_{\text{LNAME, FNAME}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$$

QUERY 4

Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

$$\text{SMITHS}(\text{ESSN}) \leftarrow \pi_{\text{SSN}}(\sigma_{\text{LNAME='Smith'}}(\text{EMPLOYEE}))$$

$$\text{SMITH\_WORKER\_PROJ} \leftarrow \pi_{\text{PNO}}(\text{WORKS\_ON} * \text{SMITHS})$$

$$\text{MGRS} \leftarrow \pi_{\text{LNAME, DNUMBER}}(\text{EMPLOYEE} \bowtie_{\text{SSN=MGRSSN}} \text{DEPARTMENT})$$

$$\text{SMITH\_MANAGED\_DEPTS}(\text{DNUM}) \leftarrow \pi_{\text{DNUMBER}}(\sigma_{\text{LNAME='Smith'}}(\text{MGRS}))$$

$$\text{SMITH\_MGR\_PROJS}(\text{PNO}) \leftarrow \pi_{\text{PNUMBER}}(\text{SMITH\_MANAGED\_DEPTS} * \text{PROJECT})$$

$$\text{RESULT} \leftarrow (\text{SMITH\_WORKER\_PROJS} \cup \text{SMITH\_MGR\_PROJS})$$

QUERY 5

List the names of all employees with two or more dependents.

Strictly speaking, this query cannot be done in the *basic relational algebra*. We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* DEPENDENT_NAME values.

$$T_1(\text{SSN, NO\_OF\_DEPTS}) \leftarrow {}_{\text{ESSN}}\mathfrak{F}_{\text{COUNT DEPENDENT\_NAME}}(\text{DEPENDENT})$$

$$T_2 \leftarrow \sigma_{\text{NO\_OF\_DEPS} \geq 2}(T_1)$$

$$\text{RESULT} \leftarrow \pi_{\text{LNAME, FNAME}}(T_2 * \text{EMPLOYEE})$$

Petitioner Uber Ex-1052, 0252

QUERY 6

Retrieve the names of employees who have no dependents.

$ALL\_EMPS \leftarrow \pi_{SSN}(EMPLOYEE)$

$EMPS\_WITH\_DEPS(SSN) \leftarrow \pi_{ESSN}(DEPENDENT)$

$EMPS\_WITHOUT\_DEPS \leftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$

$RESULT \leftarrow \pi_{LNAME, FNAME}(EMPS\_WITHOUT\_DEPS * EMPLOYEE)$

QUERY 7

List the names of managers who have at least one dependent.

$MGRS(SSN) \leftarrow \pi_{MGRSSN}(DEPARTMENT)$

$EMPS\_WITH\_DEPS(SSN) \leftarrow \pi_{ESSN}(DEPENDENT)$

$MGRS\_WITH\_DEPS \leftarrow (MGRS \cap EMPS\_WITH\_DEPS)$

$RESULT \leftarrow \pi_{LNAME, FNAME}(MGRS\_WITH\_DEPS * EMPLOYEE)$

As we mentioned earlier, the same query can in general be specified in many different ways. For example, the operations can often be applied in various sequences. In addition, some operations can be used to replace others; for example, the INTERSECTION operation in Query 7 can be replaced by a NATURAL JOIN. As an exercise, try to do each of the above example queries using different operations.[15] In Chapters 8 and 9 we will show how these queries are written in other relational languages.

7.7 Summary

In this chapter we presented the modeling concepts provided by the relational model of data. We also discussed the relational algebra and additional operations that can be used to manipulate relations. We started by introducing the concepts of domains, attributes, and tuples. We then defined a relation schema as a list of attributes that describe the structure of a relation. A relation, or relation state, is a set of tuples that conform to the schema.

Several characteristics differentiate relations from ordinary tables or files. The first is that tuples in a relation are not ordered. The second involves the ordering of attributes in a relation schema and the corresponding ordering of values within a tuple. We gave an alternative definition of relation that does not require these two orderings, but we continued to use the first definition, which requires attributes and tuple values to be ordered, for convenience. We then discussed values in tuples and introduced null values to represent missing or unknown information.

15. When queries are optimized (see Chapter 18), the system will choose a particular sequence of operations that corresponds to an execution strategy that can be executed efficiently.

Petitioner Uber Ex-1052, 0253

We then discussed the relational model constraints, starting with domain constraints, then key constraints, including the concepts of superkey, candidate key, and primary key, and the NOT NULL constraint on attributes. We then defined relational databases and relational database schemas. Additional relational constraints include the entity integrity constraint, which prohibits primary key attributes from being null. The interrelation constraint of referential integrity was then described, which is used to maintain consistency of references among tuples from different relations.

The modification operations on the relational model are Insert, Delete, and Update. Each operation may violate certain types of constraints. Whenever an operation is applied, the database state after the operation is executed must be checked to ensure that no constraints are violated.

We then described the basic relational algebra, which is a set of operations for manipulating relations that can be used to specify queries. We presented the various operations and illustrated the types of queries for which each is used. Table 7.1 lists the various relational algebra operations we discussed. The unary relational operators SELECT and PROJECT, as well as the RENAME operation, were discussed first. Then we discussed binary set theoretic operations requiring that relations on which they are applied be union compatible; these include UNION, INTERSECTION, and SET DIFFERENCE. The CARTESIAN PRODUCT operation is another set operation that can be used to combine tuples from two relations, producing all possible combinations. We showed how CARTESIAN PRODUCT followed by SELECT can identify related tuples from two relations. The JOIN operations can directly identify and combine related tuples. Join operations include THETA JOIN, EQUIJOIN, and NATURAL JOIN.

We then discussed some important types of queries that *cannot* be stated with the basic relational algebra operations. We introduced the AGGREGATE FUNCTION operation to deal with aggregate types of requests. We discussed recursive queries and showed how some types of recursive queries can be specified. We then presented the OUTER JOIN and OUTER UNION operations, which extend JOIN and UNION.

Review Questions

7.1. Define the following terms: *domain*, *attribute*, *n-tuple*, *relation schema*, *relation state*, *degree of a relation*, *relational database schema*, *relational database state*.

7.2. Why are tuples in a relation not ordered?

7.3. Why are duplicate tuples not allowed in a relation?

7.4. What is the difference between a key and a superkey?

7.5. Why do we designate one of the candidate keys of a relation to be the primary key?

7.6. Discuss the characteristics of relations that make them different from ordinary tables and files.

7.7. Discuss the various reasons that lead to the occurrence of null values in relations.

7.8. Discuss the entity integrity and referential integrity constraints. Why is each considered important?

7.9. Define *foreign key*. What is this concept used for? How does it play a role in the join operation?

Petitioner Uber Ex-1052, 0254

Table 7.1 Operations of Relational Algebra

Operation	Purpose	Notation
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{<\text{join condition}>} R_2$, or $R_1 \bowtie_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{<\text{join condition}>} R_2$, or $R_1 *_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$ or $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation R(X) that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

7.10. Discuss the various update operations on relations and the types of integrity constraints that must be checked for each update operation.
7.11. List the operations of relational algebra and the purpose of each.
7.12. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?
7.13. Discuss some types of queries for which renaming of attributes is necessary in order to specify the query unambiguously.
7.14. Discuss the various types of JOIN operations. Why is theta join required?
7.15. What is the FUNCTION operation? What is it used for?

Petitioner Uber Ex-1052, 0255

7.16. How are the OUTER JOIN operations different from the (inner) JOIN operations? How is the OUTER UNION operation different from UNION?

Exercises

7.17. Show the result of each of the example queries in Section 7.6 as it would apply to the database of Figure 7.6.

7.18. Specify the following queries on the database schema shown in Figure 7.5, using the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database of Figure 7.6.

a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the 'ProductX' project.
b. List the names of all employees who have a dependent with the same first name as themselves.
c. Find the names of all employees who are directly supervised by 'Franklin Wong'.
d. For each project, list the project name and the total hours per week (by all employees) spent on that project.
e. Retrieve the names of all employees who work on every project.
f. Retrieve the names of all employees who do not work on any project.
g. For each department, retrieve the department name and the average salary of all employees working in that department.
h. Retrieve the average salary of all female employees.
i. Find the names and addresses of all employees who work on at least one project located in Houston but whose department has no location in Houston.
j. List the last names of all department managers who have no dependents.

7.19. Suppose that each of the following update operations is applied directly to the database of Figure 7.7. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

a. Insert <'Robert', 'F', 'Scott', '943775543', '1952-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.
b. Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.
c. Insert <'Production', 4, '943775543', '1998-10-01'> into DEPARTMENT.
d. Insert <'677678989', null, '40.0'> into WORKS_ON.
e. Insert <'453453453', 'John', M, '1970-12-12', 'SPOUSE'> into DEPENDENT.
f. Delete the WORKS_ON tuples with ESSN = '333445555'.
g. Delete the EMPLOYEE tuple with SSN = '987654321'.
h. Delete the PROJECT tuple with PNAME = 'ProductX'.
i. Modify the MGRSSN and MGRSTARTDATE of the DEPARTMENT tuple with DNUMBER = 5 to '123456789' and '1999-10-01', respectively.
j. Modify the SUPERSSN attribute of the EMPLOYEE tuple with SSN = '999887777' to '943775543'.
k. Modify the HOURS attribute of the WORKS_ON tuple with ESSN = '999887777' and PNO = 10 to '5.0'.

Petitioner Uber Ex-1052, 0256

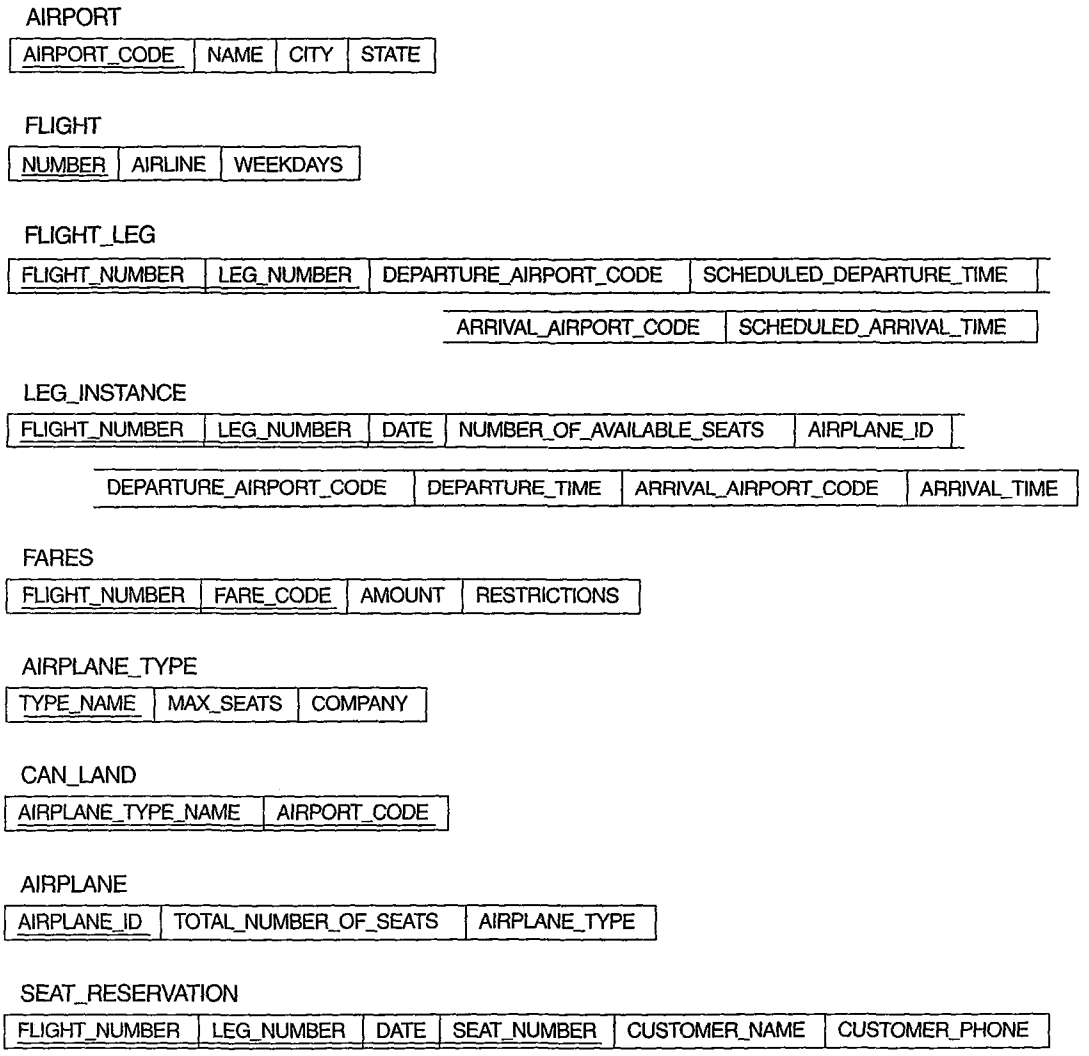


Figure 7.19 The AIRLINE relational database schema.

7.20. Consider the AIRLINE relational database schema shown in Figure 7.19, which describes a database for airline flight information. Each FLIGHT is identified by a flight NUMBER, and consists of one or more FLIGHT_LEGS with LEG_NUMBERS 1, 2, 3, etc. Each leg has scheduled arrival and departure times and airports and has many LEG_INSTANCES—one for each DATE on which the flight travels. FARES are kept for each flight. For each leg instance, SEAT_RESERVATIONS are kept, as are the AIRPLANE used on the leg and the actual arrival and departure times and airports. An AIRPLANE is identified by an AIRPLANE_ID and is of a particular AIRPLANE_TYPE. CAN_LAND relates

Petitioner Uber Ex-1052, 0257

AIRPLANE_TYPES to the AIRPORTS in which they can land. An AIRPORT is identified by an AIRPORT_CODE. Specify the following queries in relational algebra:

a. For each flight, list the flight number, the departure airport for the first leg of the flight, and the arrival airport for the last leg of the flight.
b. List the flight numbers and weekdays of all flights or flight legs that depart from Houston Intercontinental Airport (airport code 'IAH') and arrive in Los Angeles International Airport (airport code 'LAX').
c. List the flight number, departure airport code, scheduled departure time, arrival airport code, scheduled arrival time, and weekdays of all flights or flight legs that depart from some airport in the city of Houston and arrive at some airport in the city of Los Angeles.
d. List all fare information for flight number 'CO197'.
e. Retrieve the number of available seats for flight number 'CO197' on '1999-10-09'.

7.21. Consider an update for the AIRLINE database to enter a reservation on a particular flight or flight leg on a given date.

a. Give the operations for this update.
b. What types of constraints would you expect to check?
c. Which of these constraints are key, entity integrity, and referential integrity constraints, and which are not?
d. Specify all the referential integrity constraints on Figure 7.19.

7.22. Consider the relation

CLASS(Course#, Univ_Section#, InstructorName, Semester, BuildingCode, Room#, TimePeriod, Weekdays, CreditHours).

This represents classes taught in a university, with unique Univ_Section#. Identify what you think should be various candidate keys, and write in your own words the constraints under which each candidate key would be valid.

7.23. Consider the LIBRARY relational schema shown in Figure 7.20, which is used to keep track of books, borrowers, and book loans. Referential integrity constraints are shown as directed arcs in Figure 7.20, as in the notation of Figure 7.7. Write down relational expressions for the following queries on the LIBRARY database:

a. How many copies of the book titled *The Lost Tribe* are owned by the library branch whose name is 'Sharpstown'?
b. How many copies of the book titled *The Lost Tribe* are owned by each library branch?
c. Retrieve the names of all borrowers who do not have any books checked out.
d. For each book that is loaned out from the 'Sharpstown' branch and whose DueDate is today, retrieve the book title, the borrower's name, and the borrower's address.
e. For each library branch, retrieve the branch name and the total number of books loaned out from that branch.
f. Retrieve the names, addresses, and number of books checked out for all borrowers who have more than five books checked out.

Petitioner Uber Ex-1052, 0258

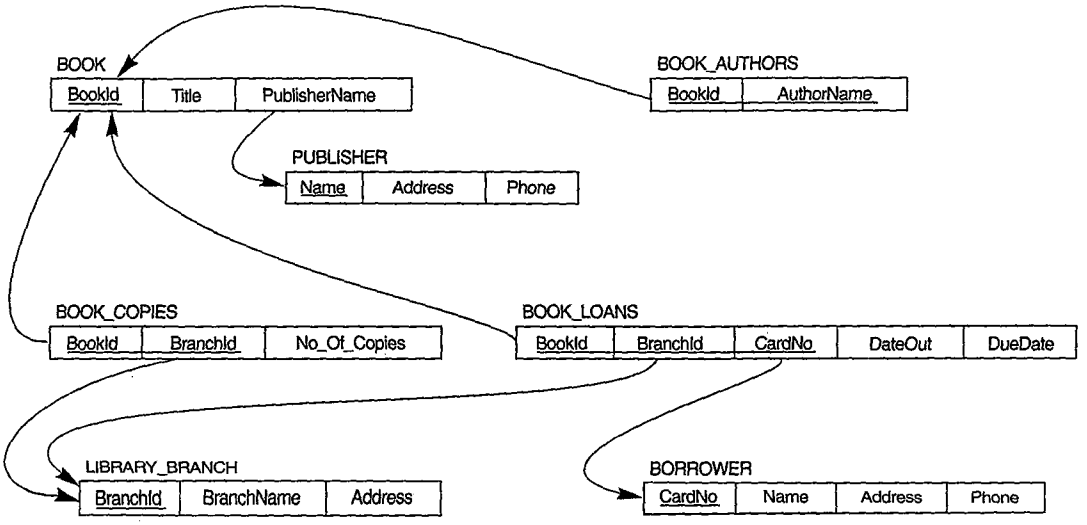


Figure 7.20 A relational database schema for a LIBRARY database.

g. For each book authored (or coauthored) by 'Stephen King,' retrieve the title and the number of copies owned by the library branch whose name is 'Central.'

7.24. Consider the following six relations for an order processing database application in a company:

CUSTOMER(Cust#, Cname, City)

ORDER(Order#, Odate, Cust#, Ord_Amt)

ORDER_ITEM(Order#, Item#, Qty)

ITEM(Item#, Unit_price)

SHIPMENT(Order#, Warehouse#, Ship_date)

WAREHOUSE(Warehouse#, City)

Here, Ord_Amt refers to total dollar amount of an order; Odate is the date the order was placed; Ship_date is the date an order is shipped from the warehouse. Assume that an order can be shipped from several warehouses. Specify the foreign keys for the above schema, stating any assumptions you make. Then specify the following queries in relational algebra:

a. List the Order# and Ship_date for all orders shipped from Warehouse number 'W2'.

b. List the Warehouse information from which the Customer named 'Jose Lopez' was supplied his orders. Produce a listing: Order#, Warehouse#.

c. Produce a listing: CUSTNAME, #OFORDERS, AVG_ORDER_AMT, where the middle column is the total number of orders by the customer and the last column is the average order amount for that customer.

Petitioner Uber Ex-1052, 0259

d. List the orders that were not shipped within 30 days of ordering.
e. List the Order# for orders that were shipped from *all* warehouses that the company has in New York.

7.25. Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:

SALESPERSON(SSN, Name, Start_Year, Dept_No)

TRIP(SSN, From_City, To_City, Departure_Date, Return_Date, Trip_ID)

EXPENSE(Trip_ID, Account#, Amount)

Specify the foreign keys for the above schema, stating any assumptions you make. Then specify the following queries in relational algebra:

a. Give the details (all attributes of TRIP relation) for trips that exceeded $2000 in expenses.
b. Print the SSN of salesman who took trips to 'Honolulu'.
c. Print the total trip expenses incurred by the salesman with SSN = '234-56-7890'.

7.26. Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(SSN, Name, Major, Bdate)

COURSE(Course#, Cname, Dept)

ENROLL(SSN, Course#, Quarter, Grade)

BOOK_ADOPTION(Course#, Quarter, Book_ISBN)

TEXT(Book_ISBN, Book_Title, Publisher, Author)

Specify the foreign keys for the above schema, stating any assumptions you make. Then specify the following queries in relational algebra:

a. List the number of courses taken by all students named 'John Smith' in Winter 1999 (i.e., Quarter = 'W99').
b. Produce a list of textbooks (include Course#, Book_ISBN, Book_Title) for courses offered by the 'CS' department that have used more than two books.
c. List any department that has *all* its adopted books published by 'BC Publishing'.

7.27. Consider the two tables T1 and T2 shown in Figure 7.21. Show the results of the following operations:

a. $T1 \bowtie_{T1.P = T2.A} T2$
b. $T1 \bowtie_{T1.Q = T2.B} T2$
c. $T1 \overline{\bowtie}_{T1.P = T2.A} T2$
d. $T1 \overline{\bowtie}_{T1.Q = T2.B} T2$
e. $T1 \cup T2$
f. $T1 \bowtie_{(T1.P = T2.A \text{ AND } T1.R = T2.C)} T2$

7.28. Consider the following relations for a database that keeps track of auto sales in a car dealership (Option refers to some optional equipment installed on an auto):

CAR(Serial-No, Model, Manufacturer, Price)

Petitioner Uber Ex-1052, 0260

Table T1

P	Q	R
10	a	5
15	b	8
25	a	6

Table T2

A	B	C
10	b	6
25	c	3
10	b	5

Figure 7.21 A database state for the relations T1 and T2.

OPTIONS(Serial-No, Option-Name, Price)

SALES(Salesperson-id, Serial-No, Date, Sale-price)

SALESPERSON(Salesperson-id, Name, Phone)

First, specify the foreign keys for the above schema, stating any assumptions you make. Next, populate the relations with a few example tuples, and then show an example of an insertion in the SALES and SALESPERSON relations that *violates* the referential integrity constraints and another insertion that does not. Then specify the following queries in relational algebra:

a. For the salesperson named 'Jane Doe', list the following information for all the cars she sold: Serial#, Manufacturer, Sale-price.
b. List the Serial# and Model of cars that have no options.
c. Consider the natural join operation between SALESPERSON and SALES. What is the meaning of a left outer join for these tables (do not change the order of relations). Explain with an example.
d. Write a query in relational algebra involving selection and one set operation and say in words what the query does.

Selected Bibliography

The relational model was introduced by Codd (1970) in a classic paper. Codd also introduced relational algebra and laid the theoretical foundations for the relational model in a series of papers (Codd 1971, 1972, 1972a, 1974); he was later given the Turing award, the highest honor of the ACM, for his work on the relational model. In a later paper, Codd (1979) discussed extending the relational model to incorporate more meta-data and semantics about the relations; he also proposed a three-valued logic to deal with uncertainty in relations and incorporating NULLs in the relational algebra. The resulting model is known as RM/T. Childs (1968) had earlier used set theory to model databases. More recently, Codd (1990) published a book examining over 300 features of the relational data model and database systems.

Since Codd's pioneering work, much research has been conducted on various aspects of the relational model. Todd (1976) describes an experimental DBMS called PRTV that directly implements the relational algebra operations. Date (1983a) discusses outer joins.

Petitioner Uber Ex-1052, 0261

Schmidt and Swenson (1975) introduces additional semantics into the relational model by classifying different types of relations. Chen's (1976) Entity Relationship model, which was discussed in Chapter 3, was a means to communicate the real-world semantics of a relational database at the conceptual level. Wiederhold and Elmasri (1979) introduces various types of connections between relations to enhance its constraints. Work on extending relational operations is discussed by Carlis (1986) and Ozsoyoglu et al. (1985). Cammarata et al. (1989) extends the relational model integrity constraints and joins. Extensions of the relational model are discussed in Chapter 13. Additional bibliographic notes for other aspects of the relational model and its languages, systems, extensions, and theory are given in Chapters 8, 9, 10, 13, 14, 15, 18, 22, 23, and 24.

Petitioner Uber Ex-1052, 0262

CHAPTER 8 SQL–The Relational Database Standard

The SQL language may be considered one of the major reasons for the success of relational databases in the commercial world. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, network or hierarchical systems—to relational systems. The reason is that even if the user became dissatisfied with the particular relational DBMS product they chose to use, converting to another relational DBMS would not be expected to be too expensive and time consuming, since both systems would follow the same language standards. In practice, of course, there are many differences between various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if both relational systems faithfully support the standard, then conversion between the two systems should be much simplified. Another advantage of having such a standard is that users may write statements in a database application program that can access data stored in two or more relational DBMSs without having to change the database sub-language (SQL) if both relational DBMSs support standard SQL.

This chapter presents the main features of the SQL standard for *commercial relational DBMSs*, whereas Chapter 7 presented the most important *formalisms* underlying the relational data model. In Chapter 7 we discussed the relational algebra operations; these operations are very important for understanding the types of requests that may be specified on a

Petitioner Uber Ex-1052, 0263

relational database. They are also important for query processing and optimization in a relational DBMS, as we shall see in Chapter 18. However, the relational algebra operations are considered to be too technical for most commercial DBMS users. One reason is because a query in relational algebra is written as a sequence of operations that, when executed, produce the required result. Hence, the user must specify how—that is, *in what order*—to execute the query operations. On the other hand, the SQL language provides a high-level *declarative* language interface, so the user only specifies *what* the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS. SQL includes some features from relational algebra, but it is based to a greater extent on the *tuple relational calculus*, which is another formal query language for relational databases that we shall describe in Section 9.3. The SQL syntax is more user-friendly than either of the two formal languages.

The name SQL is derived from Structured Query Language. Originally, SQL was called SEQUEL (for *S*tructured *E*nglish *QUE*ry *L*anguage) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. A joint effort by ANSI (the American National Standards Institute) and ISO (the International Standards Organization) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1. A revised and much expanded standard called SQL2 (also referred to as SQL-92) has subsequently been developed. Plans are already well underway for SQL3, which will further extend SQL with object-oriented and other recent database concepts.

SQL is a comprehensive database language; it has statements for data definition, query, and update. Hence, it is both a DDL *and* a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as C or PASCAL.[1] We will discuss most of these topics in the following subsections. In our discussion, we will mostly follow SQL2. Features of SQL3 are overviewed in Section 13.4.

Section 8.1 describes the SQL2 DDL commands for creating and modifying schemas, tables, and constraints. Sections 8.2 describes the basic SQL constructs for specifying retrieval queries and Section 8.3 goes over more complex features. Section 8.4 describes the SQL commands for inserting, deleting and updating, and Section 8.5 discusses the concept of views (virtual tables). Section 8.6 shows how general constraints may be specified as assertions or triggers. Section 8.7 lists some SQL features that are presented in other chapters of the book; these include embedded SQL in Chapter 10, transaction control in Chapter 19, and security/authorization in Chapter 22. Section 8.8 summarizes the chapter.

For the reader who desires a less comprehensive introduction to SQL, parts or all of the following sections may be skipped: 8.2.5, 8.3, 8.5, 8.6, and 8.7.

1. Originally, SQL had statements for creating and dropping indexes on the files that represent relations, but these have been dropped from the current SQL2 standard.

Petitioner Uber Ex-1052, 0264

8.1 Data Definition, Constraints, and Schema Changes in SQL2

SQL uses the terms **table**, **row**, and **column** for relation, tuple, and attribute, respectively. We will use the corresponding terms interchangeably. The SQL2 commands for data definition are CREATE, ALTER, and DROP; these are discussed in Sections 8.1.2 to 8.1.4. First, however, we discuss schema and catalog concepts in Section 8.1.1. Section 8.1.2 describes how tables are created, the available data types for attributes, and how constraints are specified. Sections 8.1.3 and 8.1.4 describe the **schema evolution commands** available in SQL2, which can be used to alter the schema by adding or dropping tables, attributes, and constraints. We only give an overview of the most important features. Details can be found in the SQL2 document.

8.1.1 Schema and Catalog Concepts in SQL2

Early versions of SQL did not include the concept of a relational database schema; all tables (relations) were considered part of the same schema. The concept of an SQL schema was incorporated into SQL2 in order to group together tables and other constructs that belong to the same database application. An SQL **schema** is identified by a **schema name**, and includes an authorization identifier to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema. Schema **elements** include the tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier JSMITH:

CREATE SCHEMA COMPANY AUTHORIZATION JSMITH;

In addition to the concept of schema, SQL2 uses the concept of **catalog**—a named collection of schemas in an SQL environment. A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the element descriptors of all the schemas in the catalog to authorized users. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as domain definitions.

8.1.2 The CREATE TABLE Command and SQL2 Data Types and Constraints

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified—within the CREATE TABLE statement—after the attributes are

Petitioner Uber Ex-1052, 0265

declared, or they can be added later using the ALTER TABLE command (see Section 8.1.4). Figure 8.1(a) shows sample data definition statements in SQL for the relational database schema shown in Figure 7.7. Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing:

CREATE TABLE COMPANY.EMPLOYEE ...

rather than

CREATE TABLE EMPLOYEE ...

as in Figure 8.1(a), we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.

Data Types and Domains in SQL2. The **data types** available for attributes include numeric, character-string, bit-string, date, and time. **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT), and real numbers of various precision (FLOAT, REAL, DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j)—where i, the *precision*, is the total number of decimal digits and j, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

Character-string data types are either fixed-length—CHAR(n) or CHARACTER(n), where n is the number of characters—or varying-length—VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters. **Bit-string** data types are either of fixed length n—BIT(n)—or varying length—BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is one.

There are new data types for **date** and **time** in SQL2. The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY typically in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND, typically in the form HH:MM:SS. Only valid dates and times should be allowed by the SQL implementation. In addition, a data type TIME(i), where i is called *time fractional seconds precision*, specifies i + 1 additional positions for TIME—one position for an additional separator character, and i positions for specifying decimal fractions of a second. A TIME WITH TIME ZONE data type includes an additional six positions for specifying the *displacement* from the standard universal time zone, which is in the range +13:00 to −12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session. Finally, a **timestamp** data type (TIMESTAMP) includes both the DATE and TIME fields, plus a minimum of six positions for fractions of seconds and an optional WITH TIME ZONE qualifier.

Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

In SQL2, it is possible to specify the data type of each attribute directly, as in Figure 8.1(a); alternatively, a domain can be declared, and the domain name used. This makes it

Petitioner Uber Ex-1052, 0266

```
CREATE TABLE EMPLOYEE
       ( FNAME              VARCHAR(15)        NOT NULL,
         MINIT              CHAR,
         LNAME              VARCHAR(15)        NOT NULL,
         SSN                CHAR(9)            NOT NULL,
         BDATE              DATE,
         ADDRESS            VARCHAR(30),
         SEX                CHAR,
         SALARY             DECIMAL(10,2),
         SUPERSSN           CHAR(9),
         DNO                INT                NOT NULL,
       PRIMARY KEY (SSN),
       FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
       FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE DEPARTMENT
       ( DNAME              VARCHAR(15)        NOT NULL,
         DNUMBER            INT                NOT NULL,
         MGRSSN             CHAR(9)            NOT NULL,
         MGRSTARTDATE       DATE,
       PRIMARY KEY (DNUMBER),
       UNIQUE (DNAME),
       FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) );
CREATE TABLE DEPT_LOCATIONS
       ( DNUMBER            INT                NOT NULL,
         DLOCATION          VARCHAR(15)        NOT NULL,
       PRIMARY KEY (DNUMBER, DLOCATION),
       FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE PROJECT
       ( PNAME              VARCHAR(15)        NOT NULL,
         PNUMBER            INT                NOT NULL,
         PLOCATION          VARCHAR(15),
         DNUM               INT                NOT NULL,
       PRIMARY KEY (PNUMBER),
       UNIQUE (PNAME),
       FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER) );
CREATE TABLE WORKS_ON
       ( ESSN               CHAR(9)            NOT NULL,
         PNO                INT                NOT NULL,
         HOURS              DECIMAL(3,1)       NOT NULL,
       PRIMARY KEY (ESSN, PNO),
       FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN),
       FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER) );
CREATE TABLE DEPENDENT
       ( ESSN               CHAR(9)            NOT NULL,
         DEPENDENT_NAME     VARCHAR(15)        NOT NULL,
         SEX                CHAR,
         BDATE              DATE,
         RELATIONSHIP       VARCHAR(8),
       PRIMARY KEY (ESSN, DEPENDENT_NAME),
       FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) );
```

Figure 8.1(a) An illustration of SQL2 data definition. SQL2 statements defining the COMPANY schema of Figure 7.7.

Petitioner Uber Ex-1052, 0267

easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE AS CHAR(9);

We can use SSN_TYPE in place of CHAR(9) in Figure 8.1(a) for the attributes SSN and SUPERSSN of EMPLOYEE, MGRSSN of DEPARTMENT, ESSN of WORKS_ON, and ESSN of DEPENDENT. A domain can also have an optional default specification via a DEFAULT clause, as we will discuss later for attributes.

Specifying Constraints and Default Values in SQL2. Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute. This should always be specified for the primary key attributes of each relation, as well as for any other attributes whose values are required not to be NULL, as shown in Figure 8.1(a). It is also possible to define a *default value* for an attribute by appending the clause DEFAULT <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute. Figure 8.1(b) illustrates an example of specifying a default manager for a new department and a default department for a new employee. If no default clause is specified, the default *default value* (!) is NULL.

Following the attribute (or column) specifications, additional *table constraints* can be specified on a table, including keys and referential integrity, as illustrated in Figure 8.1(a).[2] The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. The UNIQUE clause specifies alternate (or secondary) keys. Referential integrity is specified via the FOREIGN KEY clause.

As we discussed in Section 7.2.4, a referential integrity constraint can be violated when tuples are inserted or deleted or when a foreign key attribute value is modified. In SQL2, the schema designer can specify the action to be taken if a referential integrity constraint is violated upon deletion of a referenced tuple or upon modification of a referenced primary key value, by attaching a **referential triggered action** clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. We illustrate this with the example shown in Figure 8.1(b). Here, the database designer chooses SET NULL ON DELETE and CASCADE ON UPDATE for the foreign key SUPERSSN of EMPLOYEE. This means that if the tuple for a supervising employee is *deleted*, the value of SUPERSSN is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple. On the other hand, if the SSN value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to SUPERSSN for all employee tuples referencing the updated employee tuple.

2. Key and referential integrity constraints were not included in earlier versions of SQL. In some earlier implementations, keys were specified implicitly at the internal level via the CREATE INDEX command.

Petitioner Uber Ex-1052, 0268

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE or ON UPDATE; the value of the affected referencing attributes is changed to NULL for SET NULL, and to the specified default value for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the foreign key to the updated (new) primary key value for all referencing tuples. It is the responsibility of the database designer to choose the appropriate action and to specify it in the DDL. As a general rule, the CASCADE option is suitable for "relationship" relations such as WORKS_ON, for relations that represent multivalued attributes such as DEPT_LOCATIONS, and for relations that represent weak entity types such as DEPENDENT.

Figure 8.1(b) also illustrates how a constraint may be given a name, following the keyword CONSTRAINT. The names of all constraints within a particular schema must be

```
CREATE TABLE EMPLOYEE
      (...,
       DNO              INT    NOT NULL    DEFAULT 1,
      CONSTRAINT EMPPK
       PRIMARY KEY (SSN),
      CONSTRAINT EMPSUPERFK
       FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
                    ON DELETE SET NULL    ON UPDATE CASCADE,
      CONSTRAINT EMPDEPTFK
       FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
                    ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPARTMENT
      (...,
       MGRSSN   CHAR(9) NOT NULL DEFAULT '888665555',
       ...,
       CONSTRAINT DEPTPK
        PRIMARY KEY (DNUMBER),
       CONSTRAINT DEPTSK
        UNIQUE (DNAME),
       CONSTRAINT DEPTMGRFK
        FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN)
             ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPT_LOCATIONS
      (...,
       PRIMARY KEY (DNUMBER, DLOCATION),
       FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
        ON DELETE CASCADE    ON UPDATE CASCADE );
```

Figure 8.1(b) An illustration of SQL2 data definition. Specifying default attribute values and referential triggered actions.

Petitioner Uber Ex-1052, 0269

unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint, as we shall discuss in Section 8.1.4. Giving names to constraints is optional.

The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from **virtual relations,** created through the CREATE VIEW statement (see Section 8.5), which may or may not correspond to an actual physical file. In SQL the attributes in a base table are considered to be *ordered in the sequence in which they are specified* in the CREATE TABLE statement. However, rows (tuples) are not considered to be ordered within a relation.

8.1.3 The DROP SCHEMA and DROP TABLE Commands

If a whole schema is not needed any more, the DROP SCHEMA command can be used. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY CASCADE;

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed.

If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 7.6, we can get rid of the DEPENDENT relation by issuing the command:

DROP TABLE DEPENDENT CASCADE;

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views (see Section 8.5). With the CASCADE option, all such constraints and views that reference the table are dropped automatically from the schema, along with the table itself.

8.1.4 The ALTER TABLE Command

The definition of a base table can be changed by using the ALTER TABLE command, which is a **schema evolution** command. The possible *alter table actions* include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relations in the COMPANY schema, we can use the command:

ALTER TABLE COMPANY.EMPLOYEE ADD JOB VARCHAR(12);

We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command (see Section 8.4). If no default clause is specified, the new attribute will have NULLs in all

Petitioner Uber Ex-1052, 0270

the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints reference the column. For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE **DROP** ADDRESS **CASCADE;**

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

ALTER TABLE COMPANY.DEPARTMENT **ALTER** MGRSSN **DROP DEFAULT;**

ALTER TABLE COMPANY.DEPARTMENT **ALTER** MGRSSN **SET DEFAULT** "333445555";

Finally, one can change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 8.1(b) from the EMPLOYEE relation, we write

ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK **CASCADE;**

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD keyword followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed in Section 8.1.2.

The preceding subsections gave an overview of the data definition and schema evolution commands of SQL2. There are many other details and options, and we refer the interested reader to the SQL and SQL2 documents listed in the bibliographical notes. The next two sections discuss the querying capabilities of SQL.

8.2 Basic Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT statement.** The SELECT statement *has no relationship* to the SELECT operation of relational algebra, which was discussed in Chapter 7. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually. We will use example queries specified on the schema of Figure 7.5 and will refer to the sample database state shown in Figure 7.6 to show the results of some of the example queries.

Before proceeding, we must point out an important distinction between SQL and the formal relational model discussed in Chapter 7: SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an SQL table is not a *set of tuples*, because a set does not allow two identical members; rather it is

Petitioner Uber Ex-1052, 0271

a **multiset** (sometimes called a *bag*) of tuples. Some SQL relations are constrained to be sets because a key constraint has been declared or because the DISTINCT option has been used with the SELECT statement (described later in this section). We should be aware of this distinction as we discuss the examples.

8.2.1 The SELECT-FROM-WHERE Structure of SQL Queries

The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block,** is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

SELECT <attribute list>
FROM <table list>
WHERE <condition>;

where:

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

We now illustrate the basic SELECT statement with some example queries. We will label the queries here with the same query numbers that appear in Chapters 7 and 9 for easy cross reference.

QUERY 0

Retrieve the birthdate and address of the employee(s) whose name is 'John B. Smith'[3]

```
Q0: SELECT  BDATE, ADDRESS
    FROM    EMPLOYEE
    WHERE   FNAME='John' AND MINIT='B' AND LNAME='Smith';
```

This query involves only the EMPLOYEE relation listed in the FROM-clause. The query *selects* the EMPLOYEE tuples that satisfy the condition of the WHERE-clause, then *projects* the result on the BDATE and ADDRESS attributes listed in the SELECT-clause. Q0 is similar to the following relational algebra expression—except that duplicates, if any, would not be eliminated:

$$\pi_{\text{BDATE,ADDRESS}}(\sigma_{\text{FNAME='John' AND MINIT='B' AND LNAME='Smith'}}(\text{EMPLOYEE}))$$

3. This is a new query that did not appear in Chapter 7.

Petitioner Uber Ex-1052, 0272

Hence, a simple SQL query with a single relation name in the FROM-clause is similar to a SELECT-PROJECT pair of relational algebra operations. The SELECT-clause of SQL specifies the *projection attributes*, and the WHERE-clause specifies the *selection condition*. The only difference is that in the SQL query we may get duplicate tuples in the result of the query, because the constraint that a relation is a set is not enforced. Figure 8.2(a) shows the result of query Q0 on the database of Figure 7.6.

(a)

BDATE	ADDRESS
1965-01-09	731 Fondren, Houston, TX

(b)

FNAME	LNAME	ADDRESS
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

(c)

PNUMBER	DNUM	LNAME	ADDRESS	BDATE
10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

(d)

E.FNAME	E.LNAME	S.FNAME	S.LNAME
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

(e)

SSN
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

SSN	DNAME
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

(g)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

Figure 8.2 Results of queries specified on the database of Figure 7.6. (a) Result of Q0. (b) Result of Q1. (c) Result of Q2. (d) Result of Q8. (e) Result of Q9. (f) Result of Q10. (g) Result of Q1C.

Petitioner Uber Ex-1052, 0273

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT  FNAME, LNAME, ADDRESS
    FROM    EMPLOYEE, DEPARTMENT
    WHERE   DNAME='Research' AND DNUMBER=DNO;
```

Query Q1 is similar to a SELECT–PROJECT–JOIN sequence of relational algebra operations. Such queries are often called **select–project–join queries.** In the WHERE-clause of Q1, the condition DNAME = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra. The condition DNUMBER = DNO is a **join condition,** which corresponds to a JOIN condition in the relational algebra. The result of query Q1 is shown in Figure 8.2(b). In general, any number of select and join conditions may be specified in a single SQL query. The next example is a select–project–join query with *two* join conditions.

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
Q2: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
    FROM    PROJECT, DEPARTMENT, EMPLOYEE
    WHERE   DNUM=DNUMBER AND MGRSSN=SSN AND
            PLOCATION='Stafford';
```

The join condition DNUM = DNUMBER relates a project to its controlling department, whereas the join condition MGRSSN = SSN relates the controlling department to the employee who manages that department. The result of query Q2 is shown in Figure 8.2(c).

8.2.2 Dealing with Ambiguous Attribute Names and Renaming (Aliasing)

In SQL the same name can be used for two (or more) attributes as long as the attributes are in *different relations*. If this is the case, and a query refers to two or more attributes with the same name, we must **qualify** the attribute name with the relation name, to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figures 7.5 and 7.6 the DNO and LNAME attributes of the EMPLOYEE relation were called DNUMBER and NAME and the DNAME attribute of DEPARTMENT was also called NAME; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes NAME and DNUMBER in Q1A to

Petitioner Uber Ex-1052, 0274

specify which ones we are referring to, because the attribute names are used in both relations:

```
Q1A: SELECT  FNAME, EMPLOYEE.NAME, ADDRESS
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DEPARTMENT.NAME='Research' AND
             DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

Ambiguity also arises in the case of queries that refer to the same relation twice, as in the following example.

QUERY 8

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.[4]

```
Q8: SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM    EMPLOYEE AS E, EMPLOYEE AS S
    WHERE   E.SUPERSSN=S.SSN;
```

In this case, we are allowed to declare alternative relation names E and S, called **aliases** or **tuple variables,** for the EMPLOYEE relation. An alias can follow the keyword AS, as shown above in Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the WHERE-clause of Q8. It is also possible to rename the relation attributes within the query in SQL2 by giving them aliases; for example, if we write

```
EMPLOYEE AS E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)
```

in the FROM-clause, FN becomes an alias for FNAME, MI for MINIT, LN for LNAME, and so on. In Q8, we can think of E and S as two *different copies* of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees; and the second, S, represents employees in the role of supervisors. We can now join the two copies. Of course, in reality there is *only one* EMPLOYEE relation, and the join condition is meant to join the relation with itself by matching the tuples that satisfy the join condition E.SUPERSSN = S.SSN. Notice that this is an example of a one-level recursive query, as we discussed in Section 7.5.2. As in relational algebra, we *cannot* specify a general recursive query, with an unknown number of levels, in a single SQL2 statement.[5]

The result of query Q8 is shown in Figure 8.2(d). Whenever one or more aliases are given to a relation, we can use these names to represent different references to that relation. This permits multiple references to the same relation within a query. Notice that, if we want to, we can use this alias-naming mechanism in any SQL query, whether or not the same relation needs to be referenced more than once. For example, we could specify query

4. This query did not appear in Chapter 7; hence it is given the number 8 to distinguish it from queries 1 through 7 of Section 7.6.
5. A construct for specifying recursive queries is planned for SQL3.

Petitioner Uber Ex-1052, 0275

Q1A as in Q1B just for convenience to shorten the relation names that prefix the attributes:

```
Q1B: SELECT  E.FNAME, E.NAME, E.ADDRESS
     FROM    EMPLOYEE E, DEPARTMENT D
     WHERE   D.NAME='Research' AND D.DNUMBER=E.DNUMBER;
```

8.2.3 Unspecified WHERE-Clause and Use of Asterisk (*)

We discuss two more features of SQL here. A *missing WHERE-clause* indicates no condition on tuple selection; hence, *all tuples* of the relation specified in the FROM-clause qualify and are selected for the query result.[6] If more than one relation is specified in the FROM-clause and there is no WHERE-clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected. For example, Query 9 selects all EMPLOYEE SSNs (Figure 8.2e), and Query 10 selects all combinations of an EMPLOYEE SSN and a DEPARTMENT DNAME (Figure 8.2f).

QUERIES 9 and 10

Select all EMPLOYEE SSNs (Q9), and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME (Q10) in the database.

```
Q9:  SELECT  SSN
     FROM    EMPLOYEE;

Q10: SELECT  SSN, DNAME
     FROM    EMPLOYEE, DEPARTMENT;
```

It is extremely important to specify every selection and join condition in the WHERE-clause; if any such condition is overlooked, incorrect and very large relations may result. Notice that Q10 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra. If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the CROSS PRODUCT.

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes*. For example, query Q1C retrieves all the attribute values of EMPLOYEE tuples who work in DEPARTMENT number 5 (Figure 8.2g); query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT he or she works in for every employee of the 'Research' department; and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
Q1C: SELECT  *
     FROM    EMPLOYEE
     WHERE   DNO=5;
```

6. This is equivalent to the condition WHERE TRUE, which means *every row in the table is selected.*

Petitioner Uber Ex-1052, 0276

```
Q1D:  SELECT  *
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNAME='Research' AND DNO=DNUMBER;
Q10A: SELECT  *
      FROM    EMPLOYEE, DEPARTMENT;
```

8.2.4 Tables as Sets in SQL

As we mentioned earlier, SQL usually treats a table not as a set but rather as a **multiset;** *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function (see Section 8.3.5) is applied to tuples, in most cases we do not want to eliminate duplicates.

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.[7] If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword DISTINCT in the SELECT-clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates whereas a query with SELECT ALL does not (specifying SELECT with neither ALL nor DISTINCT is equivalent to SELECT ALL). For example, Query 11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in Figure 8.3(a). If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword DISTINCT as in Q11A we accomplish this, as shown in Figure 8.3(b).

QUERY 11

Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:  SELECT ALL      SALARY
      FROM            EMPLOYEE;
Q11A: SELECT DISTINCT SALARY
      FROM            EMPLOYEE;
```

7. In general, an SQL table is not required to have a key although in most cases there will be one.

Petitioner Uber Ex-1052, 0277

(a)

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

(b)

SALARY
30000
40000
25000
43000
38000
55000

(c)

FNAME	LNAME

(d)

FNAME	LNAME
James	Borg

Figure 8.3 Results of some other queries specified on the database shown in Figure 7.6. (a) Result of Q11. (b) Result of Q11A. (c) Result of Q16. (d) Result of Q18.

SQL has directly incorporated some of the set operations of relational algebra. There is a set union operation (UNION), and in SQL2 there are also set difference (EXCEPT) and set intersection (INTERSECT) operations.[8] The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result*. Because these set operations apply only to *union-compatible relations*, we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

QUERY 4

Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4: (SELECT   DISTINCT PNUMBER
     FROM     PROJECT, DEPARTMENT, EMPLOYEE
     WHERE    DNUM=DNUMBER AND MGRSSN=SSN AND
              LNAME='Smith')
     UNION
     (SELECT  DISTINCT PNUMBER
     FROM     PROJECT, WORKS_ON, EMPLOYEE
     WHERE    PNUMBER=PNO AND ESSN=SSN AND LNAME='Smith');
```

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a

8. SQL2 also has corresponding multiset operations, which are followed by the keyword ALL (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets.

Petitioner Uber Ex-1052, 0278

'Smith' as a worker on the project. Notice that, if several employees have the last name 'Smith', the project names involving any of them will be retrieved. Applying the UNION operation to the two SELECT queries gives the desired result.

8.2.5 Substring Comparisons, Arithmetic Operators, and Ordering

In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. Partial strings are specified by using two reserved characters: '%' replaces an arbitrary number of characters, and the underscore (_) replaces a single character. For example, consider the following query.

QUERY 12

Retrieve all employees whose address is in Houston, Texas.

```
Q12:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   ADDRESS LIKE '%Houston,TX%';
```

To retrieve all employees who were born during the 1950s, we can use Query 26. Here, '5' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore[9] serving as a placeholder for an arbitrary character.

QUERY 12A

Find all employees who were born during the 1950s.

```
Q12A: SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   BDATE LIKE'_ _ 5 _ _ _ _ _ _ _';
```

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (−), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue Query 13 to see what their salaries would become.

9. If underscore or % are literal characters in the string, they should be preceded with an *escape character*, which is specified after the string; for example, 'AB_CD\%EF' ESC '\' represents the literal string 'AB_CD%EF'.

Petitioner Uber Ex-1052, 0279

QUERY 13

Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
Q13:  SELECT   FNAME, LNAME, 1.1*SALARY
      FROM     EMPLOYEE, WORKS_ON, PROJECT
      WHERE    SSN=ESSN AND PNO=PNUMBER AND
               PNAME='ProductX';
```

For string data types, the concatenate operator '| |' can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing ('+') or decrementing ('–') a date, time, or timestamp by a type-compatible interval. In addition, an interval value can be specified as the difference between two date, time, or timestamp values. Another comparison operator that can be used for convenience is BETWEEN, which is illustrated in Query 14.[10]

QUERY 14

Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT   *
      FROM     EMPLOYEE
      WHERE    (SALARY BETWEEN 30000 AND 40000) AND DNO = 5;
```

SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the ORDER BY-clause. This is illustrated by Query 15.

QUERY 15

Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.

```
Q15:  SELECT   DNAME, LNAME, FNAME, PNAME
      FROM     DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
      WHERE    DNUMBER=DNO AND SSN=ESSN AND
               PNO=PNUMBER
      ORDER BY DNAME, LNAME, FNAME;
```

The default order is in ascending order of values. We can specify the keyword DESC if we want a descending order of values. The keyword ASC can be used to specify ascending

10. The condition (SALARY BETWEEN 30000 AND 40000) is equivalent to ((SALARY $\geq$ 30000) AND (SALARY $\leq$ 40000)).

Petitioner Uber Ex-1052, 0280

order explicitly. If we want descending order on DNAME and ascending order on LNAME, FNAME, the ORDER BY-clause of Q15 becomes

```
ORDER BY  DNAME DESC, LNAME ASC, FNAME ASC
```

8.3 More Complex SQL Queries

In the previous section, we described the basic types of queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex queries. We discuss several of these features in this section.

8.3.1 Nested Queries and Set Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries,** which are complete SELECT . . . FROM . . . WHERE . . . blocks within the WHERE-clause of another query. That other query is called the **outer query.** Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A:

```
Q4A:  SELECT   DISTINCT PNUMBER
      FROM     PROJECT
      WHERE    PNUMBER IN  (SELECT  PNUMBER
                           FROM    PROJECT, DEPARTMENT,
                                   EMPLOYEE
                           WHERE   DNUM=DNUMBER AND
                                   MGRSSN=SSN AND
                                   LNAME='Smith')
               OR
               PNUMBER IN  (SELECT  PNO
                           FROM    WORKS_ON, EMPLOYEE
                           WHERE   ESSN=SSN AND
                                   LNAME='Smith');
```

The first nested query selects the project numbers of projects that have a 'Smith' involved as manager, while the second selects the project numbers of projects that have a 'Smith' involved as worker. In the outer query, we select a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query. The comparison operator IN compares a value v with a set (or multiset) of values V and evaluates to TRUE if v is one of the elements in V.

The IN operator can also compare a tuple of values in parentheses with a set or multiset of union-compatible tuples. For example, the query:

```
SELECT DISTINCT ESSN
FROM   WORKS_ON
```

Petitioner Uber Ex-1052, 0281

```
WHERE (PNO, HOURS) IN (SELECT PNO, HOURS FROM WORKS_ON
                       WHERE ESSN='123456789');
```

will select the social security numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose SSN = '123456789') works on.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to *some value* in the set V and is hence equivalent to IN. The keywords ANY and SOME have the same meaning. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than *all* the values in the set V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > ALL (SELECT SALARY FROM EMPLOYEE WHERE
        DNO=5);
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—once in a relation in the FROM-clause of the *outer query*, and the other in a relation in the FROM-clause of the *nested query*. The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query.** For example, in the SELECT-clause and WHERE-clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM-clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we can specify and refer to an *alias* for that relation. These rules are similar to scope rules for program variables in a programming language such as PASCAL, which allows nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16, whose result is shown in Figure 8.3(c).

QUERY 16

Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

```
Q16:  SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   E.SSN IN (SELECT  ESSN
                        FROM    DEPENDENT
                        WHERE   E.FNAME=DEPENDENT_NAME
                                AND E.SEX=SEX);
```

Petitioner Uber Ex-1052, 0282

In the nested query of Q16, we must qualify E.SEX because it refers to the SEX attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called SEX. All unqualified references to SEX in the nested query refer to SEX of DEPENDENT. However, we do not *have to* qualify FNAME and SSN because the DEPENDENT relation does not have attributes called FNAME and SSN, so there is no ambiguity.

Correlated Nested Queries. Whenever a condition in the WHERE-clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated.** We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query.* For example, we can think of Q16 as follows: for *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the ESSN values for all DEPENDENT tuples with the same sex and name as the EMPLOYEE tuple; if the SSN value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested SELECT . . . FROM . . . WHERE . . . blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q16 may be written as in Q16A:

```
Q16A: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E, DEPENDENT AS D
      WHERE   E.SSN=D.ESSN AND E.SEX=D.SEX AND
              E.FNAME=D.DEPENDENT_NAME;
```

The original SQL implementation on SYSTEM R also had a CONTAINS comparison operator, which is used to compare two sets or multisets. This operator was subsequently dropped from the language, possibly because of the difficulty in implementing it efficiently. Most commercial implementations of SQL do *not* have this operator. The CONTAINS operator compares two sets of values and returns TRUE if one set contains all values in the other set. Query 3 illustrates the use of the CONTAINS operator.

QUERY 3

Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

```
Q3:   SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   ( (SELECT    PNO
                 FROM      WORKS_ON
                 WHERE     SSN=ESSN)
                CONTAINS
                (SELECT    PNUMBER
                 FROM      PROJECT
                 WHERE     DNUM=5) );
```

Petitioner Uber Ex-1052, 0283

In Q3, the second nested query (which is not correlated with the outer query) retrieves the project numbers of all projects controlled by department 5. For *each* employee tuple, the first nested query (which is correlated) retrieves the project numbers on which the employee works; if these contain all projects controlled by department 5, the employee tuple is selected and the name of that employee is retrieved. Notice that the CONTAINS comparison operator is similar in function to the DIVISION operation of the relational algebra, described in Section 7.4.7. Because the CONTAINS operation is not part of SQL, we use the EXISTS function to specify these types of queries, as will be shown in Section 8.3.2.

8.3.2 The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. We illustrate the use of EXISTS—and also NOT EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS. This is shown as Q16B:

```
Q16B: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   EXISTS (SELECT  *
                      FROM    DEPENDENT
                      WHERE   E.SSN=ESSN AND E.SEX=SEX AND
                              E.FNAME=DEPENDENT_NAME);
```

EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query. In Q16B, the nested query references the SSN, FNAME, and SEX attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows: for each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same social security number, sex, and name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. In general, EXISTS(Q) returns TRUE if there is *at least one tuple* in the result of query Q, and it returns FALSE otherwise. On the other hand, NOT EXISTS(Q) returns TRUE if there are *no tuples* in the result of query Q, and it returns FALSE otherwise. Next, we illustrate the use of NOT EXISTS.

QUERY 6

Retrieve the names of employees who have no dependents.

```
Q6:   SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   NOT EXISTS  (SELECT  *
                           FROM    DEPENDENT
                           WHERE   SSN=ESSN);
```

Petitioner Uber Ex-1052, 0284

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected. We can explain Q6 as follows: for *each* EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose ESSN value matches the EMPLOYEE SSN; *if* the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its FNAME and LNAME. There is another SQL function UNIQUE(Q) that returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE.

QUERY 7

List the names of managers who have at least one dependent.

```
Q7:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   EXISTS  (SELECT  *
                      FROM    DEPENDENT
                      WHERE   SSN=ESSN)
             AND
             EXISTS  (SELECT  *
                      FROM    DEPARTMENT
                      WHERE   SSN=MGRSSN);
```

One way to write this query is shown in Q7, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exist, we select the EMPLOYEE tuple. Can you rewrite this query using only a single nested query or no nested queries?

Query 3, which we used to illustrate the CONTAINS comparison operator, can be stated using EXISTS and NOT EXISTS in SQL systems. There are two options. The first is to use the well known set theory transformation that *(S1 CONTAINS S2)* is logically equivalent to *(S2 EXCEPT S1) is empty;*[11] this is shown as Q3A.

```
Q3A: SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   NOT EXISTS
             (   (SELECT  PNUMBER
                  FROM    PROJECT
                  WHERE   DNUM=5)
                  EXCEPT
                  (SELECT  PNO
                  FROM    WORKS_ON
                  WHERE   SSN=ESSN) );
```

11. Recall that EXCEPT is the set difference operator.

Petitioner Uber Ex-1052, 0285

The second option is shown as Q3B below. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3, which used the CONTAINS comparison operator, and Q3A, which uses NOT EXISTS and EXCEPT. However, CONTAINS is not part of SQL, and not all relational systems have the EXCEPT operator even though it is part of SQL2:

```
Q3B:  SELECT   LNAME, FNAME
      FROM     EMPLOYEE
      WHERE    NOT EXISTS
               (SELECT *
               FROM     WORKS_ON B
               WHERE    (B.PNO IN    (SELECT   PNUMBER
                                      FROM     PROJECT
                                      WHERE    DNUM=5) )
                        AND
                        NOT EXISTS (SELECT   *
                                    FROM     WORKS_ON C
                                    WHERE    C.ESSN=SSN
                                             AND
                                             C.PNO=B.PNO) );
```

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose PNO is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same PNO and the same SSN as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: select each employee such that there does not exist a project controlled by department 5 that the employee does not work on.

Notice that Query 3 is typically stated in relational algebra by using the DIVISION operation. Moreover, Query 3 requires a type of quantifier called a **universal quantifier** in the relational calculus (see Section 9.3.5). The negated existential quantifier NOT EXISTS can be used to express a universally quantified query, as we shall discuss in Chapter 9.

8.3.3 Explicit Sets and NULLS in SQL

We have seen several queries with a nested query in the WHERE-clause. It is also possible to use an **explicit set of values** in the WHERE-clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

QUERY 17

Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

Petitioner Uber Ex-1052, 0286

```
Q17:  SELECT  DISTINCT ESSN
      FROM    WORKS_ON
      WHERE   PNO IN (1, 2, 3);
```

SQL allows queries that check whether a value is NULL—missing or undefined or not applicable. However, rather than using = or ≠ to compare an attribute to NULL, SQL uses IS or IS NOT. This is because SQL considers each null value as being distinct from every other null value, so equality comparison is not appropriate. It follows that, when a join condition is specified, tuples with null values for the join attributes are not included in the result (unless it is an OUTER JOIN; see Section 8.3.4). Query 18 illustrates this; its result is shown in Figure 8.3(d).

QUERY 18

Retrieve the names of all employees who do not have supervisors.

```
Q18:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   SUPERSSN IS NULL;
```

8.3.4 Renaming Attributes and Joined Tables

It is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses. For example, Q8A below shows how query Q8 can be slightly changed to retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME. The new names will appear as column headers in the query result:

```
Q8A:  SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS
              SUPERVISOR_NAME
      FROM    EMPLOYEE AS E, EMPLOYEE AS S
      WHERE   E.SUPERSSN=S.SSN;
```

The concept of a **joined table** (or **joined relation**) was incorporated into SQL2 to permit users to specify a table resulting from a join operation *in the FROM-clause* of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE-clause. For example, consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier first to specify the join of the EMPLOYEE and DEPARTMENT relations, and then to select the desired tuples and attributes. This can be written in SQL2 as in Q1A:

```
Q1A:  SELECT  FNAME, LNAME, ADDRESS
      FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
      WHERE   DNAME='Research';
```

Petitioner Uber Ex-1052, 0287

The FROM-clause in Q1A contains a single *joined table*. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations R and S, no join condition is specified; an implicit equi-join condition for *each pair of attributes with the same name* from R and S is created. Each such pair of attributes is included only once in the resulting relation (see Section 7.4.5.)

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as DNAME, DNO (to match the name of the desired join attribute DNO in EMPLOYEE), MSSN, and MSDATE. The implied join condition for this NATURAL JOIN is EMPLOYEE.DNO = DEPT.DNO, because this is the only pair of attributes with the same name after renaming:

```
Q1B:  SELECT  FNAME, LNAME, ADDRESS
      FROM    (EMPLOYEE NATURAL JOIN
              (DEPARTMENT AS DEPT (DNAME, DNO, MSSN,
              MSDATE)))
      WHERE   DNAME='Research;
```

The default type of join in a joined table is an **inner** join, where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query Q8A, only employees that *have a supervisor* are included in the result; an EMPLOYEE tuple whose value for SUPERSSN is NULL is excluded. If the user requires that all employees be included, an outer join must be used explicitly (see Section 7.5.3 for a definition of OUTER JOIN). In SQL2, this is handled by explicitly specifying the OUTER JOIN in a joined table, as illustrated in Q8B:

```
Q8B:  SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS
              SUPERVISOR_NAME
      FROM    (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
                    ON E.SUPERSSN=S.SSN);
```

The options available for specifying joined tables in SQL2 include INNER JOIN (same as JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. In the latter three, the keyword OUTER may be omitted. It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This is illustrated by Q2A, which is a different way of specifying query Q2, using the concept of a joined table:

```
Q2A:  SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
      FROM    ((PROJECT JOIN DEPARTMENT ON DNUM=
                    DNUMBER) JOIN EMPLOYEE ON MGRSSN=SSN)
      WHERE   PLOCATION='Stafford';
```

Petitioner Uber Ex-1052, 0288

8.3.5 Aggregate Functions and Grouping

In Section 7.5.1, we introduced the concept of an aggregate function as a relational operation. Because grouping and aggregation are required in many database applications, SQL has features that incorporate these concepts. The first of these is a number of built-in functions: COUNT, SUM, MAX, MIN, and AVG. The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG are applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT-clause or in a HAVING-clause (which we will introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.[12] We illustrate the use of these functions with example queries.

QUERY 19

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19:  SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
              AVG (SALARY)
      FROM    EMPLOYEE;
```

If we want to get the preceding function values for employees of a specific department—say the 'Research' department—we can write Query 20, where the EMPLOYEE tuples are restricted by the WHERE-clause to those employees who work for the 'Research' department.

QUERY 20

Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:  SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
              AVG (SALARY)
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNO=DNUMBER AND DNAME='Research';
```

QUERIES 21 and 22

Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

12. Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, DATE, TIME and TIMESTAMP domains have total orderings on their values, as do alphabetic strings.

Petitioner Uber Ex-1052, 0289

```
Q21:  SELECT  COUNT (*)
      FROM    EMPLOYEE;
Q22:  SELECT  COUNT (*)
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNO=DNUMBER AND DNAME='Research';
```

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

QUERY 23

Count the number of distinct salary values in the database.

```
Q23:  SELECT  COUNT (DISTINCT SALARY)
      FROM    EMPLOYEE;
```

Notice that, if we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, we *get the same result* as COUNT(*) because duplicate values will not be eliminated, and so the number of values will be the same as the number of tuples.[13] The preceding examples show how functions are applied to retrieve a summary value from the database. In some cases we may need to use functions to select particular tuples. In such cases we specify a correlated nested query with the desired function, and we use that nested query in the WHERE-clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents (Query 5), we can write:

```
Q5:   SELECT  LNAME, FNAME
      FROM    EMPLOYEE
      WHERE   (SELECT COUNT (*)
               FROM   DEPENDENT
               WHERE  SSN=ESSN) >= 2;
```

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to 2, the employee tuple is selected.

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation*, based on some attribute values. For example, we may want to find the average salary of employees in each department or the number of employees who work on each project. In these cases we need to group the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**, and we need to apply the function to each such group independently. SQL has a **GROUP BY-clause** for this purpose. The GROUP BY-clause specifies the grouping attributes, which should *also appear in the SELECT-clause*, so that the value resulting from applying each function to a group of tuples appears along with the value of the grouping attribute(s).

13. Unless some tuples have NULL for SALARY, in which case they are not counted.

Petitioner Uber Ex-1052, 0290

QUERY 24

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24:  SELECT    DNO, COUNT (*), AVG (SALARY)
      FROM      EMPLOYEE
      GROUP BY DNO;
```

In Q24, the EMPLOYEE tuples are divided into groups—each group having the same value for the grouping attribute DNO. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples. Figure 8.4(a) illustrates how grouping works on Q24, and it also shows the result of Q24.

QUERY 25

For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25:  SELECT    PNUMBER, PNAME, COUNT (*)
      FROM      PROJECT, WORKS_ON
      WHERE     PNUMBER=PNO
      GROUP BY PNUMBER, PNAME;
```

Q25 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations. Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions*. For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a **HAVING-clause,** which can appear in conjunction with a GROUP BY-clause, for this purpose. HAVING provides a condition on the group of tuples associated with each value of the grouping attributes; and only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

QUERY 26

For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:  SELECT    PNUMBER, PNAME, COUNT (*)
      FROM      PROJECT, WORKS_ON
      WHERE     PNUMBER=PNO
      GROUP BY PNUMBER, PNAME
      HAVING    COUNT (*) > 2;
```

Petitioner Uber Ex-1052, 0291

(a)

FNAME	MINIT	LNAME	SSN	. . .	SALARY	SUPERSSN	DNO
John	B	Smith	123456789		30000	333445555	5
Franklin		Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	. . .	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad		Jabbar	987987987		25000	987654321	4
James		Bong	888665555		55000	null	1

Grouping EMPLOYEE tuples by the value of DNO.

DNO	COUNT (*)	AVG (SALARY)
5	4	33250
4	3	31000
1	1	55000

Result of Q24.

(b)

PNAME	PNUMBER		ESSN	PNO	HOURS
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	. . .	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	null
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING.

PNAME	PNUMBER		ESSN	PNO	HOURS
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10	. . .	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	null
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition.

PNAME	COUNT (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26 (PNUMBER not shown).

Figure 8.4 Results of GROUP BY and HAVING. (a) Results for Query 24. (b) Results for Query 26.

Petitioner Uber Ex-1052, 0292

Notice that, while selection conditions in the WHERE-clause limit the *tuples* to which functions are applied, the HAVING-clause serves to choose *whole groups*. Figure 8.4(b) illustrates the use of HAVING and displays the result of Q26.

QUERY 27

For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27:  SELECT    PNUMBER, PNAME, COUNT (*)
      FROM      PROJECT, WORKS_ON, EMPLOYEE
      WHERE     PNUMBER=PNO AND SSN=ESSN AND DNO=5
      GROUP BY  PNUMBER, PNAME;
```

Here we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE-clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the function in the SELECT-clause and another to the function in the HAVING-clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT-clause. Suppose that we write the following *incorrect* query:

```
SELECT     DNAME, COUNT (*)
FROM       DEPARTMENT, EMPLOYEE
WHERE      DNUMBER=DNO AND SALARY>40000
GROUP BY   DNAME
HAVING     COUNT (*) > 5;
```

This is incorrect because it will select only departments that have more than five employees *who each earn more than $40,000*. The rule is that the WHERE-clause is executed first, to select individual tuples; the HAVING-clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than $40,000, *before* the function in the HAVING-clause is applied. One way to write the query correctly is to use a nested query, as shown in Query 28.

QUERY 28

For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:  SELECT    DNUMBER, COUNT (*)
      FROM      DEPARTMENT, EMPLOYEE
      WHERE     DNUMBER=DNO AND SALARY>40000 AND
                DNO IN (SELECT    DNO
```

Petitioner Uber Ex-1052, 0293

```
                    FROM        EMPLOYEE
                    GROUP BY    DNO
                    HAVING      COUNT (*) > 5)
        GROUP BY    DNUMBER;
```

8.3.6 Discussion and Summary of SQL Queries

A query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory. The clauses are specified in the following order, with the clauses between square brackets [. . .] being optional:

SELECT <attribute and function list>
FROM <table list>
[**WHERE** <condition>]
[**GROUP BY** <grouping attribute(s)>]
[**HAVING** <group condition>]
[**ORDER BY** <attribute list>];

The SELECT-clause lists the attributes or functions to be retrieved. The FROM-clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE-clause specifies the conditions for selection of tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

A query is evaluated *conceptually* by applying first the FROM-clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE-clause, and then GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING, ORDER BY) are specified, we can *think conceptually* of a query as being executed as follows: for *each combination of tuples*—one from each of the relations specified in the FROM-clause—evaluate the WHERE-clause; if it evaluates to TRUE, place the values of the attributes specified in the SELECT-clause from this tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each DBMS has special query optimization routines to decide on an execution plan that is efficient. We discuss query processing and optimization in Chapter 18.

In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantage is that users can choose the technique they are most comfortable with when specifying a query. For example, many queries may be specified with join conditions in the WHERE-clause, or by using joined relations in the FROM-clause, or with some form of nested queries and the IN comparison operator. Some users may be more comfortable with one approach, whereas

Petitioner Uber Ex-1052, 0294

others may be more comfortable with another. From the programmer's and the system's query optimization point of view, it is generally preferable to write a query with as little nesting and implied ordering as possible.

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: the DBMS should process the same query in the same way, regardless of how the query is specified. But this is quite difficult in practice, as each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient. Ideally, the user should worry only about specifying the query correctly. It is the responsibility of the DBMS to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others.

8.4 Insert, Delete, and Update Statements in SQL

In SQL three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

8.4.1 The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation shown in Figure 7.5 and specified in the CREATE TABLE EMPLOYEE . . . command in Figure 8.1, we can use U1:

```
U1:   INSERT INTO  EMPLOYEE
      VALUES       ('Richard', 'K', 'Marini', '653298653', '1962-12-30','98
                   Oak Forest,Katy,TX','M', 37000, '987654321', 4);
```

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes, but only a few of those attributes are assigned values in the new tuple. These attributes must include all attributes with NOT NULL specification and no default value; attributes with NULL allowed or DEFAULT values are the ones that can be *left out*. For example, to enter a tuple for a new EMPLOYEE for whom we know only the FNAME, LNAME, DNO, and SSN attributes, we can use U1A:

```
U1A:  INSERT INTO  EMPLOYEE (FNAME, LNAME, DNO, SSN)
      VALUES       ('Richard', 'Marini', 4, '653298653');
```

Petitioner Uber Ex-1052, 0295

Attributes not specified in U1A are set to their DEFAULT or to NULL, and the values are listed in the same order as the *attributes are listed in the INSERT command itself*. It is also possible to insert into a relation *multiple tuples* separated by commas in a single INSERT command. The attribute values forming *each tuple* are enclosed in parentheses.

A DBMS that fully implements SQL2 should support and enforce all the integrity constraints that can be specified in the DDL. However, some DBMSs do not incorporate all the constraints, in order to maintain the efficiency of the DBMS and because of the complexity of enforcing all constraints. If a system does not support some constraint—say, referential integrity—the users or programmers must enforce the constraint. For example, if we issue the command in U2 on the database shown in Figure 7.6, a DBMS not supporting referential integrity will do the insertion even though no DEPARTMENT tuple exists in the database with DNUMBER = 2. It is the responsibility of the user to check that any such constraints *whose checks are not implemented by the DBMS* are not violated. However, the DBMS must implement checks to enforce all the SQL integrity constraints *it supports*. A DBMS enforcing NOT NULL will reject an INSERT command in which an attribute declared to be NOT NULL does not have a value; for example, U2A would be *rejected* because no SSN value is provided.

```
U2:   INSERT INTO  EMPLOYEE (FNAME, LNAME, SSN, DNO)
      VALUES       ('Robert', 'Hatcher', '980760540', 2);
      (* U2 is rejected if referential integrity checking is provided by DBMS *)

U2A:  INSERT INTO  EMPLOYEE (FNAME, LNAME, DNO)
      VALUES       ('Robert', 'Hatcher', 5);
      (* U2A is rejected if NOT NULL checking is provided by DBMS *)
```

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query*. For example, to create a temporary table that has the name, number of employees, and total salaries for each department, we can write the statements in U3A and U3B:

```
U3A:  CREATE TABLE  DEPTS_INFO
      (DEPT_NAME    VARCHAR(15),
      NO_OF_EMPS    INTEGER,
      TOTAL_SAL     INTEGER);

U3B:  INSERT INTO   DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,
                    TOTAL_SAL)
      SELECT        DNAME, COUNT (*), SUM (SALARY)
      FROM          (DEPARTMENT JOIN EMPLOYEE ON
                    DNUMBER=DNO)
      GROUP BY      DNAME;
```

A table DEPTS_INFO is created by U3A and is loaded with the summary information retrieved from the database by the query in U3B. We can now query DEPTS_INFO as we could any other relation; and when we do not need it any more, we can remove it by using the DROP TABLE command. Notice that the DEPTS_INFO table may not be up to date;

Petitioner Uber Ex-1052, 0296

that is, if we update either the DEPARTMENT or the EMPLOYEE relations after issuing U3B, the information in DEPTS_INFO *becomes outdated*. We have to create a view (see Section 8.5) to keep such a table up to date.

8.4.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE-clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL (see Section 8.1.2). Depending on the number of tuples selected by the condition in the WHERE-clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE-clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.[14] The DELETE commands in U4A to U4D, if applied independently to the database of Figure 7.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A: DELETE FROM  EMPLOYEE
     WHERE        LNAME='Brown';

U4B: DELETE FROM  EMPLOYEE
     WHERE        SSN='123456789';

U4C: DELETE FROM  EMPLOYEE
     WHERE        DNO IN  (SELECT  DNUMBER
                           FROM    DEPARTMENT
                           WHERE   DNAME='Research');

U4D: DELETE FROM  EMPLOYEE;
```

8.4.3 The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE-clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity constraints of the DDL (see Section 8.1.2). An additional **SET-clause** specifies the attributes to be modified and their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

```
U5:  UPDATE  PROJECT
     SET     PLOCATION = 'Bellaire', DNUM = 5
     WHERE   PNUMBER=10;
```

14. We must use the DROP TABLE command to remove the table completely.

Petitioner Uber Ex-1052, 0297

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, as shown in U6. In this request, the modified SALARY value depends on the original SALARY value in each tuple, so two references to the SALARY attribute are needed. In the SET-clause, the reference to the SALARY attribute on the right refers to the old SALARY value *before modification*, and the one on the left refers to the new SALARY value *after modification*:

```
U6:  UPDATE  EMPLOYEE
     SET     SALARY = SALARY *1.1
     WHERE   DNO IN  (SELECT  DNUMBER
                      FROM    DEPARTMENT
                      WHERE   DNAME='Research');
```

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly specifies a single relation only. To modify multiple relations, we must issue several UPDATE commands. These (and other SQL commands) could be embedded in a general-purpose program, as we shall discuss in Chapter 10.

8.5 Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We then show how views are specified, and we discuss the problem of updating a view, and how a view can be implemented by the DBMS.

8.5.1 Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables.[15] These other tables could be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a **virtual table,** in contrast to base tables whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically. For example, in Figure 7.5 we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins. We can then issue queries on the view, which are specified as single-table retrievals rather than as retrievals involving two joins on three tables. We call the tables EMPLOYEE, WORKS_ON, and PROJECT the **defining tables** of the view.

15. As used here, the term *view* is more limited than the term *user views* discussed in Chapters 1 and 2, since a user view would possibly include many relations.

Petitioner Uber Ex-1052, 0298

8.5.2 Specification of Views in SQL

The command to specify a view is CREATE VIEW. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes result from applying functions or arithmetic operations, we do not have to specify attribute names for the view, as they would be the same as the names of the attributes of the defining tables in the default case. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 8.5 when applied to the database schema of Figure 7.5.

```
V1:  CREATE VIEW    WORKS_ON1
     AS  SELECT     FNAME, LNAME, PNAME, HOURS
         FROM       EMPLOYEE, PROJECT, WORKS_ON
         WHERE      SSN=ESSN AND PNO=PNUMBER;

V2:  CREATE VIEW    DEPT_INFO(DEPT_NAME, NO_OF_EMPS,
                    TOTAL_SAL)
     AS  SELECT     DNAME, COUNT (*), SUM (SALARY)
         FROM       DEPARTMENT, EMPLOYEE
         WHERE      DNUMBER=DNO
         GROUP BY   DNAME;
```

In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON. View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT-clause of the query that defines the view. We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name

WORKS_ON1

FNAME	LNAME	PNAME	HOURS

DEPT_INFO

DEPT_NAME	NO_OF_EMPS	TOTAL_SAL

Figure 8.5 Two views specified on the database schema of Figure 7.5.

Petitioner Uber Ex-1052, 0299

and first name of all employees who work on 'ProjectX', we can utilize the WORKS_ON1 view and specify the query as in QV1:

```
QV1:  SELECT  FNAME, LNAME
      FROM    WORKS_ON1
      WHERE   PNAME='ProjectX';
```

The same query would require the specification of two joins if specified on the base relations; one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism (see Chapter 22).

A view is *always up to date*; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized at the time of *view definition* but rather at the time we *specify a query* on the view. It is the responsibility of the DBMS and not the user to make sure that the view is up to date.

If we do not need a view any more, we can use the DROP VIEW command to dispose of it. For example, to get rid of the view V1, we can use the SQL statements in V1A:

```
V1A:  DROP VIEW  WORKS_ON1;
```

8.5.3 View Implementation and View Update

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested. One strategy, called **query modification,** involves modifying the view query into a query on the underlying base tables. The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time. The other strategy, called **view materialization,** involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up to date. Techniques using the concept of **incremental update** have been developed for this purpose, where it is determined what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base tables. The view is generally kept as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical view table and recompute it from scratch when future queries reference the view.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a *single table* without any *aggregate functions* can be mapped to an update on the underlying base table. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations *in multiple ways*. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME

Petitioner Uber Ex-1052, 0300

attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1:  UPDATE  WORKS_ON1
      SET     PNAME = 'ProductY'
      WHERE   LNAME='Smith' AND FNAME='John' AND
              PNAME='ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. Two possible updates (a) and (b) on the base relations corresponding to UV1 are shown here:

```
(a):  UPDATE  WORKS_ON
      SET     PNO =     (SELECT PNUMBER FROM PROJECT
                        WHERE PNAME='ProductY')
      WHERE   ESSN IN   (SELECT SSN FROM EMPLOYEE
                        WHERE LNAME='Smith' AND FNAME='John')
              AND
              PNO IN    (SELECT PNUMBER FROM PROJECT
                        WHERE PNAME='ProductX');

(b):  UPDATE  PROJECT
      SET     PNAME = 'ProductY'
      WHERE   PNAME = 'ProductX';
```

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple in place of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the effect of changing all the view tuples with PNAME = 'ProductX'.

Some view updates may not make much sense; for example, modifying the TOTAL_SAL attribute of the DEPT_INFO view does not make sense because TOTAL_SAL is defined to be the sum of the individual employee salaries. This request is shown as UV2:

```
UV2:  UPDATE  DEPT_INFO
      SET     TOTAL_SAL=100000
      WHERE   DNAME='Research';
```

A large number of updates on the underlying base relations can satisfy this view update.

A view update is feasible when only *one possible update* on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, we must have a certain procedure to choose the desired update. Some researchers have developed methods

Petitioner Uber Ex-1052, 0301

for choosing the most likely update, while other researchers prefer to have the user choose the desired update mapping during view definition.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key (or possibly some other candidate key) of the base relation, because this maps each (virtual) view tuple to a single base tuple.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL2, the clause WITH CHECK OPTION must be added at the end of the view definition if a view is to be updated. This allows the system to check for view updatability and to plan an execution strategy for view updates.

8.6 Specifying General Constraints as Assertions

In SQL2, users can specify more general constraints—those that do not fall into any of the categories described in Section 8.1.2—via **declarative assertions,** using the **CREATE ASSERTION statement** of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE-clause of an SQL query. For example, to specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for" in SQL2, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS (SELECT * FROM EMPLOYEE E, EMPLOYEE M,
                                  DEPARTMENT D
                    WHERE E.SALARY>M.SALARY AND
                          E.DNO=D.DNUMBER AND
                          D.MGRSSN=M.SSN) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE-clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of conditions. Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated.** The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.

Note that the CHECK clause and constraint condition can also be used in conjunction with the CREATE DOMAIN statement (see Section 8.1.2) to specify constraints on a particular domain, such as restricting the values of a domain to a *subrange* of the data type

Petitioner Uber Ex-1052, 0302

for the domain. For example, to restrict the values of department numbers to an integer number between 1 and 20, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER
CHECK (D_NUM > 0 AND D_NUM < 21);
```

Earlier versions of SQL had two types of statements to declare constraints: ASSERT and TRIGGER. The **ASSERT statement** is somewhat similar to CREATE ASSERTION of SQL2 with a different syntax. The **TRIGGER statement** is used in a different way. In many cases it is convenient to specify the type of action to be taken in case of a constraint violation. Rather than offering users only the option of aborting the transaction that causes a violation, the DBMS should make other options available. For example, it may be useful to specify a constraint that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user, and the constraint is thus used to **monitor** the database. Other actions may be specified, such as executing a specific procedure or triggering other updates. A mechanism called a *trigger* has been proposed to implement such actions in earlier versions of SQL. A **trigger** specifies a **condition** and an **action** to be taken in case that condition is satisfied. The condition is usually specified as an assertion that invokes or "triggers" the action when it becomes TRUE. We will discuss triggers in more detail in Chapter 23 when we describe *active databases*.

8.7 Additional Features of SQL

There are a number of additional features of SQL that we have not described in this chapter, but will discuss elsewhere in the book. These are as follows:

- SQL has language constructs for specifying the *granting and revoking of privileges* to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or UPDATE—to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called **GRANT** and **REVOKE**—are discussed in Chapter 22 where we discuss database security and authorization.
- SQL has a methodology for embedding SQL statements in a general purpose programming language, such as C, C++, COBOL, or PASCAL. SQL also has **language bindings** to various programming languages that specify the correspondence of SQL data types to the data types of each of the programming languages. **Embedded SQL** is based on the concept of a **cursor** that can range over the query result one tuple at a time. We will discuss embedded SQL, and give examples of how it is used in relational database programming, in Chapter 10.

Petitioner Uber Ex-1052, 0303

- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes. We will discuss these commands in Chapter 19.
- Each commercial DBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We called these commands a *storage definition language* (SDL) in Chapter 2. Earlier versions of SQL had commands for **creating indexes,** but these were removed from the language because they were not at the conceptual schema level (see Chapter 2). We will discuss these commands for a specific commercial relational DBMS in Chapter 10.

8.8 Summary

In this chapter we presented the SQL database language. This language or variations of it have been implemented as interfaces to many commercial relational DBMSs, including IBM's DB2 and SQL/DS, ORACLE, INGRES, INFORMIX, and SYBASE. The original version of SQL was implemented in the experimental DBMS called SYSTEM R, which was developed at IBM Research. SQL is designed to be a comprehensive language that includes statements for data definition, queries, updates, view definition, and constraint specification. We discussed each of these in separate sections of this chapter. In the final section we discussed additional features that are described elsewhere in the book. Our emphasis was on the SQL2 standard. The next version of the standard, called SQL3, is well underway. It will incorporate object-oriented and other advanced database features into the standard. We discuss some of the proposed features of SQL3 in Chapter 13.

Table 8.1 shows a summary of the syntax (or structure) of various SQL statements. This summary is not meant to be comprehensive nor to describe every possible SQL construct; rather, it is meant to serve as a quick reference to the major types of constructs available in SQL. We use BNF notation, where nonterminal symbols are shown in angled brackets < . . . >, optional parts are shown in square brackets [. . .], repetitions are shown in braces { . . . }, and alternatives are shown in parentheses (. . . | . . . | . . .).[16]

Review Questions

8.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 7? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

8.2. List the data types that are allowed for SQL2 attributes.

8.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 7? What about general integrity constraints?

16. The full syntax of SQL2 is described in a document of over 500 pages.

Petitioner Uber Ex-1052, 0304

Table 8.1 Summary of SQL Syntax

```
CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
                          {, <column name> <column type> [<attribute constraint>] }
                          [<table constraint> {,<table constraint>}])

DROP TABLE <table name>

ALTER TABLE <table name> ADD <column name> <column type>

SELECT [DISTINCT] <attribute list>
FROM (<table name> { <alias>} | <joined table>) {, (<table name> { <alias>} | <joined table>) }
[WHERE <condition>]
[GROUP BY <grouping attributes> [HAVING <group selection condition> ] ]
[ORDER BY <column name> [<order>] {, <column name> [<order>] } ]

<attribute list>::= (* | ( <column name> | <function>(([DISTINCT]<column name> | *)))
                    {,( <column name> | <function>(([DISTINCT] <column name> | *)) } ) )
<grouping attributes>::= <column name> { , <column name>}
<order>::= (ASC | DESC)

INSERT INTO <table name> [( <column name>{, <column name>} ) ]
(VALUES ( <constant value> , { <constant value>} ){,(<constant value>{,<constant value>})}
| <select statement>)

DELETE FROM <table name>
[WHERE <selection condition>]

UPDATE <table name>
SET <column name>=<value expression> { , <column name>=<value expression> }
[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name>*
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[CLUSTER]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

*These last two commands are not part of standard SQL2.

Petitioner Uber Ex-1052, 0305

8.4. What is a view in SQL, and how is it defined? Discuss the problems that may arise when one attempts to update a view. How are views typically implemented?
8.5. Describe the six clauses in the syntax of an SQL query, and show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?
8.6. Describe conceptually how an SQL query will be executed by specifying the conceptual order of executing each of the six clauses.

Exercises

8.7. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. What are the referential integrity constraints that should hold on the schema? Write appropriate SQL DDL statements to define the database.
8.8. Repeat Exercise 8.7, but use the AIRLINE database schema of Figure 7.19.
8.9. Consider the LIBRARY relational database schema of Figure 7.20. Choose the appropriate action (reject, cascade, set to null, set to default) for each referential integrity constraint, both for *delete* of a referenced tuple, and for *update* of a primary key attribute value in a referenced tuple. Justify your choices.
8.10. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 7.20. Use the referential actions chosen in Exercise 8.9.
8.11. Write SQL queries for the LIBRARY database queries given in Exercise 7.23.
8.12. How can the key and foreign key constraints be enforced by the DBMS? Is the enforcement technique you suggest difficult to implement? Can the constraint checks be executed efficiently when updates are applied to the database?
8.13. Specify the queries of Exercise 7.18 in SQL. Show the result of each query if it is applied to the COMPANY database of Figure 7.6.
8.14. Specify the following additional queries on the database of Figure 7.5 in SQL. Show the query results if each query is applied to the database of Figure 7.6.
   a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.
   b. Suppose that we want the number of *male* employees in each department rather than all employees (as in Exercise 8.14a). Can we specify this query in SQL? Why or why not?
8.15. Specify the updates of Exercise 7.19, using the SQL update commands.
8.16. Specify the following queries in SQL on the database schema of Figure 1.2.
   a. Retrieve the names of all senior students majoring in 'CS' (computer science).
   b. Retrieve the names of all courses taught by Professor King in 1998 and 1999.
   c. For each section taught by Professor King, retrieve the course number, semester, year, and number of students who took the section.
   d. Retrieve the name and transcript of each senior student (Class = 5) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.
   e. Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).

Petitioner Uber Ex-1052, 0306

f. Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.

8.17. Write SQL update statements to do the following on the database schema shown in Figure 1.2.
   a. Insert a new student <'Johnson', 25, 1, 'MATH'> in the database.
   b. Change the class of student 'Smith' to 2.
   c. Insert a new course <'Knowledge Engineering','CS4390', 3,'CS'>.
   d. Delete the record for the student whose name is 'Smith' and whose student number is 17.

8.18. Specify the following views in SQL on the COMPANY database schema shown in Figure 7.5.
   a. A view that has the department name, manager name, and manager salary for every department.
   b. A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department.
   c. A view that has project name, controlling department name, number of employees, and total hours worked per week on the project for each project.
   d. A view that has project name, controlling department name, number of employees, and total hours worked per week on the project for each project *with more than one employee working on it.*

8.19. Consider the following view DEPT_SUMMARY, defined on the COMPANY database of Figure 7.6:

```
CREATE VIEW  DEPT_SUMMARY (D, C, TOTAL_S, AVERAGE_S)
AS SELECT    DNO, COUNT (*), SUM (SALARY), AVG (SALARY)
   FROM      EMPLOYEE
   GROUP BY  DNO;
```

State which of the following queries and updates would be allowed on the view. If a query or update would be allowed, show what the corresponding query or update on the base relations would look like, and give its result when applied to the database of Figure 7.6.

a.
```
SELECT  *
FROM    DEPT_SUMMARY;
```
b.
```
SELECT  D, C
FROM    DEPT_SUMMARY
WHERE   TOTAL_S > 100000;
```
c.
```
SELECT  D, AVERAGE_S
FROM    DEPT_SUMMARY
WHERE   C > (SELECT C FROM DEPT_SUMMARY WHERE D=4);
```
d.
```
UPDATE  DEPT_SUMMARY
SET     D=3
WHERE   D=4;
```
e.
```
DELETE FROM DEPT_SUMMARY
WHERE   C > 4;
```

Petitioner Uber Ex-1052, 0307

8.20. Consider the relation schema CONTAINS(Parent_part#, Sub_part#); a tuple <P_i, P_j> in CONTAINS means that part P_i contains part P_j as a direct component. Suppose that we choose a part P_k that contains no other parts, and we want to find the part numbers of all parts that contain P_k, directly or indirectly at any level; this is a *recursive query* that requires computing the **transitive closure** of CONTAINS. Show that this query cannot be directly specified as a single SQL query. Can you suggest extensions to SQL to allow the specification of such queries?

8.21. Specify the queries and updates of Exercises 7.20 and 7.21, which refer to the AIRLINE database, in SQL.

8.22. Choose some database application that you are familiar with.
   a. Design a relational database schema for your database application.
   b. Declare your relations, using the SQL DDL.
   c. Specify a number of queries in SQL that are needed by your database application.
   d. Based on your expected use of the database, choose some attributes that should have indexes specified on them.
   e. Implement your database, if you have a DBMS that supports SQL.

8.23. Specify the answers to Exercises 7.24 through 7.28 in SQL.

Selected Bibliography

The SQL language, originally named SEQUEL, was based on the language SQUARE (Specifying Queries as Relational Expressions), described by Boyce et al. (1975). The syntax of SQUARE was modified into SEQUEL (Chamberlin and Boyce 1974) and then into SEQUEL 2 (Chamberlin et al. 1976), on which SQL is based. The original implementation of SEQUEL was done at IBM Research, San Jose, California.

Reisner (1977) describes a human factors evaluation of SEQUEL in which she found that users have some difficulty with specifying join conditions and grouping correctly. Date (1984b) contains a critique of the SQL language that points out its strengths and shortcomings. Date and Darwen (1993) describes SQL2. ANSI (1986) outlines the original SQL standard, and ANSI (1992) describes the SQL2 standard. Various vendor manuals describe the characteristics of SQL as implemented on DB2, SQL/DS, ORACLE, INGRES, INFORMIX, and other commercial DBMS products. Melton and Simon (1993) is a comprehensive treatment of SQL2. Horowitz (1992) discusses some of the problems related to referential integrity and propagation of updates in SQL2.

The question of view updates is addressed by Dayal and Bernstein (1978), Keller (1982), and Langerak (1990), among others. View implementation is discussed in Blakeley et al. (1989). Negri et al. (1991) describes formal semantics of SQL queries.

Petitioner Uber Ex-1052, 0308

CHAPTER 9 ER- and EER-to-Relational Mapping, and Other Relational Languages

This chapter discusses two topics that are not directly related but serve to round out our presentation of the relational data model. The first topic focuses on **designing a relational database schema** based on a conceptual schema design. This is the logical database design (or data model mapping) step discussed in Section 3.1 (see Figure 3.1). This relates the concepts of the Entity-Relationship (ER) and Enhanced-ER (EER) models, presented in Chapters 3 and 4, to the concepts of the relational model, presented in Chapter 7. In Sections 9.1 and 9.2, we show how a relational database schema can be created from a conceptual schema developed using the ER or EER models. Many CASE (Computer-Aided Software Engineering) tools are based on the ER or EER or other similar models, as we have discussed in Chapters 3 and 4. These computerized tools are used interactively by database designers to develop an ER or EER schema for a database application. Many tools use ER diagrams or variations to develop the schema graphically, and then automatically convert it into a relational database schema in the DDL of a specific relational DBMS.

The second topic introduces some **other relational languages** that are important. These are presented in Sections 9.3 through 9.5. We first describe another formal language for relational databases, the **relational calculus.** There are two variations of relational calculus: the tuple relational calculus is described in Section 9.3, and the domain relational calculus is described in Section 9.4. Some of the SQL query language constructs discussed in Chapter 8 are based on the tuple relational calculus. The relational calculus is a formal language, based on the branch of mathematical logic called

Petitioner Uber Ex-1052, 0309

predicate calculus.[1] In tuple relational calculus, variables range over tuples; whereas in domain relational calculus, variables range over the domains (values) of attributes. Finally, in Section 9.5, we give an overview of the QBE (Query-By-Example) language, which is a graphical user-friendly relational language based on domain relational calculus. Section 9.6 summarizes the chapter.

Sections 9.1 and 9.2 on relational database design assume that the reader is familiar with the material in Chapters 3 and 4, respectively.

9.1 Relational Database Design Using ER-to-Relational Mapping

Section 9.1.1 provides an outline of an algorithm that can map an ER schema into the corresponding relational database schema. Section 9.1.2 summarizes the correspondences between ER and relational model constructs. Section 9.2 discusses the options for mapping the EER model constructs—such as generalization/specialization and categories—into relations.

9.1.1 ER-to-Relational Mapping Algorithm

We now describe the steps of an algorithm for ER-to-relational mapping. We will use the COMPANY relational database schema, shown in Figure 7.5, to illustrate the mapping steps.

STEP 1: For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure 7.5 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT from Figure 3.2.[2] The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the attributes SUPERSSN and DNO of EMPLOYEE; MGRSSN and MGRSTARTDATE of DEPARTMENT; and DNUM of PROJECT. In our example, we choose SSN, DNUMBER, and PNUMBER as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively.

STEP 2: For each weak entity type W in the ER schema with owner entity type E, create a relation R, and include all simple attributes (or simple components of com-

1. In this chapter no familiarity with first-order predicate calculus, which deals with quantified variables and values, is assumed.

2. These are sometimes called *entity relations* because each tuple (row) represents an entity instance.

Petitioner Uber Ex-1052, 0310

posite attributes) of W as attributes of R. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. We include the primary key SSN of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we renamed it ESSN, although this is not necessary. The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT_NAME} because DEPENDENT_NAME is the partial key of DEPENDENT.

It is common to choose the propagate (CASCADE) option for the referential triggered action (see Section 8.1) on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both ON UPDATE and ON DELETE.

STEP 3: For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. Choose one of the relations—S, say—and include as foreign key in S the primary key of T. It is better to choose an entity type with *total participation* in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.

In our example, we map the 1:1 relationship type MANAGES from Figure 3.2 by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total (every department has a manager). We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it MGRSSN. We also include the simple attribute StartDate of the MANAGES relationship type in the DEPARTMENT relation and rename it MGRSTARTDATE.

Notice that an alternative mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This is appropriate when *both participations are total.*

STEP 4: For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the *N-side* of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R; this is because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S.

In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 3.2. For WORKS_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it SUPERSSN. The CONTROLS

Petitioner Uber Ex-1052, 0311

relationship is mapped to the foreign key attribute DNUM of PROJECT, which references the primary key DNUMBER of the DEPARTMENT relation.

STEP 5: For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations—as we did for 1:1 or 1:N relationship types—because of the M:N cardinality ratio.

In our example, we map the M:N relationship type WORKS_ON from Figure 3.2 by creating the relation WORKS_ON in Figure 7.5.[3] We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them PNO and ESSN, respectively. We also include an attribute HOURS in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {ESSN, PNO}.

The propagate (CASCADE) option for the referential triggered action (see Section 8.1) should be specified on the foreign keys in the relation corresponding to the relationship R, since each relationship instance has an existence dependency on each of the entities it relates. This can be used for both ON UPDATE and ON DELETE.

Notice that we can always map 1:1 or 1:N relationships in a manner similar to M:N relationships. This alternative is particularly useful when few relationship instances exist, in order to avoid null values in foreign keys. In this case, the primary key of the "relationship" relation will be *only one* of the foreign keys that reference the participating "entity" relations. For a 1:N relationship, this will be the foreign key that references the entity relation on the N-side. For a 1:1 relationship, the foreign key that references the entity relation with total participation (if any) is chosen as primary key.

STEP 6: For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, plus the primary key attribute K—as a foreign key in R—of the relation that represents the entity type or relationship type that has A as an attribute. The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.[4]

In our example, we create a relation DEPT_LOCATIONS. The attribute DLOCATION represents the multivalued attribute Locations of DEPARTMENT, while DNUMBER—as foreign key—represents the primary key of the DEPARTMENT relation. The primary key of

3. These are sometimes called *relationship relations* because each tuple (row) corresponds to a relationship instance.

4. In some cases when a multivalued attribute is composite, only some of the component attributes are required in the key of R; these attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute.

Petitioner Uber Ex-1052, 0312

DEPT_LOCATIONS is the combination of {DNUMBER, DLOCATION}. A separate tuple will exist in DEPT_LOCATIONS for each location that a department has.

The propagate (CASCADE) option for the referential triggered action (see Section 8.1) should be specified on the foreign key in the relation corresponding to the multivalued attribute for both ON UPDATE and ON DELETE.

Figure 7.5 (and 7.7) shows the relational database schema obtained through the preceding steps, and Figure 7.6 shows a sample database state. Notice that we did not discuss the mapping of n-ary relationship types (n > 2), because none exist in Figure 3.2; these are mapped in a similar way to M:N relationship types by including the following additional step in the mapping algorithm.

STEP 7: For each n-ary relationship type R, where n > 2, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E' corresponding to E (see Section 4.7). This concludes the mapping procedure.

For example, consider the relationship type SUPPLY of Figure 4.13(a). This can be mapped to the relation SUPPLY shown in Figure 9.1, whose primary key is the combination of foreign keys {SNAME, PARTNO, PROJNAME}.

The main point to note in a relational schema, in contrast to an ER schema, is that relationship types are not represented explicitly; instead, they are represented by having

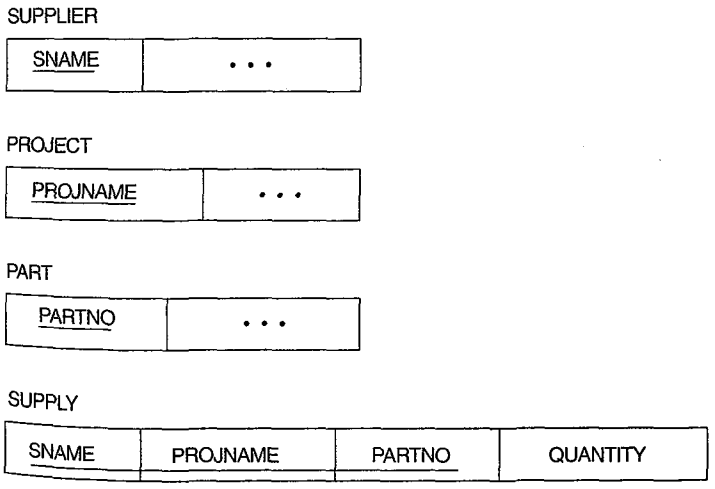


Figure 9.1 Mapping the n-ary relationship type SUPPLY from Figure 4.13(a).

Petitioner Uber Ex-1052, 0313

two attributes A and B, one a primary key and the other a foreign key—over the same domain—included in two relations S and T. Two tuples in S and T are related when they have the same value for A and B. By using the EQUIJOIN (or NATURAL JOIN) operation over S.A and T.B, we can combine all pairs of related tuples from S and T and materialize the relationship. When a binary 1:1 or 1:N relationship type is involved, a single join operation is usually needed. For a binary M:N relationship type, two join operations are needed, whereas for n-ary relationship types, n joins are needed.

For example, to form a relation that includes the employee name, project name, and hours that the employee works on each project, we need to connect each EMPLOYEE tuple to the related PROJECT tuples via the WORKS_ON relation of Figure 7.5. Hence, we must apply the EQUIJOIN operation to the EMPLOYEE and WORKS_ON relations with the join condition SSN = ESSN, and then apply another EQUIJOIN operation to the resulting relation and the PROJECT relation with join condition PNO = PNUMBER. In general, when multiple relationships need to be traversed, numerous join operations must be specified. A relational database user must always be aware of the foreign key attributes in order to use them correctly in combining related tuples from two or more relations. If an equijoin is performed among attributes of two relations that do not represent a foreign key/primary key relationship, the result can often be meaningless and may lead to spurious (invalid) data. For example, the reader can try joining PROJECT and DEPT_LOCATIONS relations on the condition DLOCATION = PLOCATION and examine the result (see also Chapter 14).

Another point to note in the relational schema is that we create a separate relation for *each* multivalued attribute. For a particular entity with a set of values for the multivalued attribute, the key attribute value of the entity is repeated once for each value of the multivalued attribute in a separate tuple. This is because the basic relational model does *not* allow multiple values (a list, or a set of values) for an attribute in a single tuple. For example, because department 5 has three locations, three tuples exist in the DEPT_LOCATIONS relation of Figure 7.6; each tuple specifies one of the locations. In our example, we apply EQUIJOIN to DEPT_LOCATIONS and DEPARTMENT on the DNUMBER attribute to get the values of all locations along with other DEPARTMENT attributes. In the resulting relation, the values of the other department attributes are repeated in separate tuples for every location that a department has.

The basic relational algebra does not have a NEST or COMPRESS operation that would produce from the DEPT_LOCATIONS relation of Figure 7.6 a set of tuples of the form {<1, Houston>, <4, Stafford>, <5, {Bellaire, Sugarland, Houston}>}. This is a serious drawback of the basic normalized or "flat" version of the relational model. On this score, the object-oriented, hierarchical, and network models have better facilities than does the relational model. The nested relational model (see Section 13.6) attempts to remedy this.

9.1.2 Summary of Mapping for Model Constructs and Constraints

We now summarize the correspondences between ER and relational model constructs and constraints in Table 9.1.

Petitioner Uber Ex-1052, 0314

Table 9.1 Correspondence between ER and Relational Models

ER Model	Relational Model
Entity type	"Entity" relation
1:1 or 1:N relationship type	Foreign key (or "relationship" relation)
M:N relationship type	"Relationship" relation and two foreign keys
n-ary relationship type	"Relationship" relation and n foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

9.2 Mapping EER Model Concepts to Relations

We now discuss the mapping of EER model concepts to relations by extending the ER-to-relational mapping algorithm that was presented in Section 9.1 above.

9.2.1 Superclass/Subclass Relationships and Specialization (or Generalization)

There are several options for mapping a number of subclasses that together form a specialization (or alternatively, that are generalized into a superclass), such as the {SECRETARY, TECHNICIAN, ENGINEER} subclasses of EMPLOYEE in Figure 4.4. We can add a further step to our ER-to-relational mapping algorithm from Section 9.1, which has seven steps, to handle the mapping of specialization. Step 8, which follows, gives the most common options; other mappings are also possible. We then discuss the conditions under which each option should be used. We use Attrs(R) to denote the attributes of relation R and PK(R) to denote the primary key of R.

STEP 8: Convert each specialization with m subclasses $\{S_1, S_2, \ldots, S_m\}$ and (generalized) superclass C, where the attributes of C are $\{k, a_1, \ldots, a_n\}$ and k is the (primary) key, into relation schemas using one of the four following options:

Option 8A: Create a relation L for C with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\}$ and PK(L) = k. Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes $Attrs(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $PK(L_i) = k$.

Option 8B: Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes $Attrs(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \ldots, a_n\}$ and $PK(L_i) = k$.

Petitioner Uber Ex-1052, 0315

Option 8C: Create a single relation L with attributes Attrs(L) = {k, a_1, . . ., a_n} ∪ {attributes of S_1} ∪ . . . ∪ {attributes of S_m} ∪ {t} and PK(L) = k. This option is for a specialization whose subclasses are *disjoint*, and t is a **type** (or **discriminating**) attribute that indicates the subclass to which each tuple belongs, if any. This option has the potential for generating a large number of null values.

Option 8D: Create a single relation schema L with attributes Attrs(L) = {k, a_1, . . ., a_n} ∪ {attributes of S_1} ∪ . . . ∪ {attributes of S_m} ∪ {t_1, t_2, . . ., t_m} and PK(L) = k. This option is for a specialization whose subclasses are *overlapping* (but will also work for a disjoint specialization), and each t_i, $1 \leq i \leq m$, is a Boolean attribute indicating whether a tuple belongs to subclass S_i.

Options 8A and 8B can be called the *multiple relation options*, whereas options 8C and 8D can be called the *single relation options*. Option 8A creates a relation L for the superclass C and its attributes, plus a relation L_i for each subclass S_i; each L_i includes the specific (or local) attributes of S_i, plus the primary key of the superclass C, which is propagated to L_i and becomes its primary key. An EQUIJOIN operation on the primary key between any L_i and L produces all the specific and inherited attributes of the entities in S_i. This option is illustrated in Figure 9.2(a) for the EER schema in Figure 4.4. Option 8A

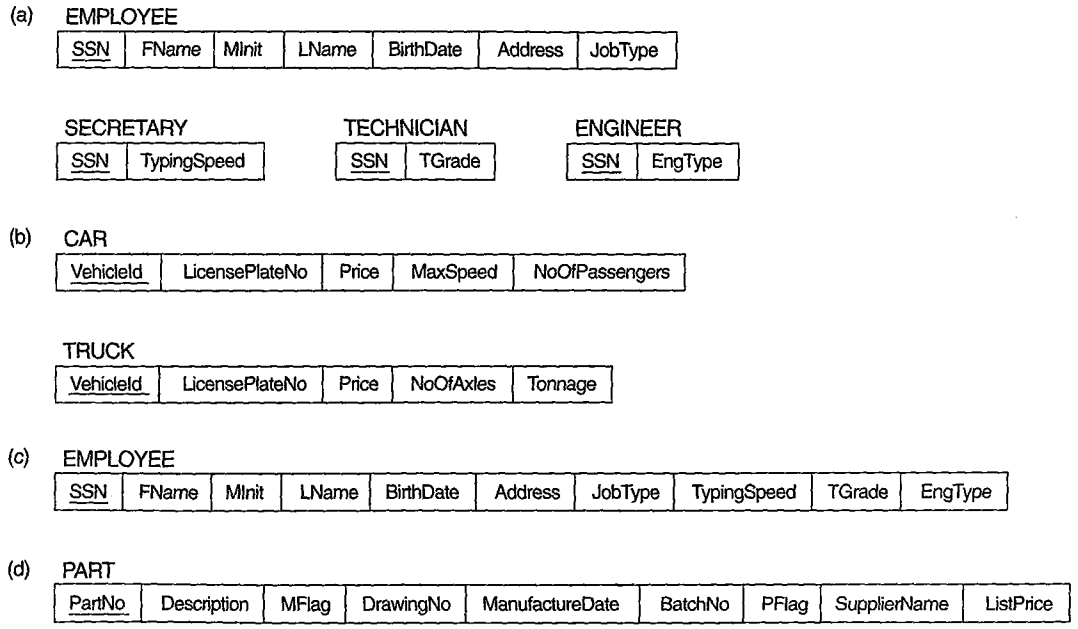


Figure 9.2 Options for mapping specializations (or generalizations) to relations. (a) Mapping the EER schema of Figure 4.4 to relations by using Option A. (b) Mapping the EER schema of Figure 4.3(b) into relations by using Option B. (c) Mapping the EER schema of Figure 4.4 by using Option C, with JobType playing the role of type attribute. (d) Mapping the EER schema of Figure 4.5 by using Option D, with two Boolean type fields Mflag and Pflag.

Petitioner Uber Ex-1052, 0316

works for any constraints on the specialization: disjoint or overlapping, total or partial. Notice that the constraint

$$\pi_{<K>}(L_i) \subseteq \pi_{<k>}(L)$$

must hold for each L_i. This specifies an *inclusion dependency* $L_i.k<L.k$ (see Section 15.4).

In option 8B, the EQUIJOIN operation is *built into* the schema and the relation L is done away with, as illustrated in Figure 9.2(b) for the EER specialization in Figure 4.3(b). This option works well only when *both* the disjoint and total constraints hold. If the specialization is not total, an entity that does not belong to any of the subclasses S_i is lost. If the specialization is not disjoint, an entity belonging to more than one subclass will have its inherited attributes from the superclass C stored redundantly in more than one L_i. With option 8B, no relation holds all the entities in the superclass C; consequently, we must apply an OUTER UNION (or FULL OUTER JOIN) operation to the L_i relations to retrieve all the entities in C. The result of the outer union will be similar to the relations under options 8C and 8D except that the type fields will be missing. Whenever we search for an arbitrary entity in C, we must search all the m relations L_i.

Options 8C and 8D create a single relation to represent the superclass C and all its subclasses. An entity that does not belong to some of the subclasses will have null values for the specific attributes of these subclasses. These options are hence not recommended if many specific attributes are defined for the subclasses. If few specific subclass attributes exist, however, these mappings are preferable to options 8A and 8B because they do away with the need to specify EQUIJOIN and OUTER UNION operations and hence can yield a more efficient implementation. Option 8C is used to handle disjoint subclasses by including a single **type** (or **image** or **discriminating**) **attribute** t to indicate the subclass to which each tuple belongs; hence, the domain of t could be {1, 2, . . ., m}. If the specialization is partial, t can have null values in tuples that do not belong to any subclass. If the specialization is attribute-defined, that attribute serves the purpose of t and t is not needed; this option is illustrated in Figure 9.2(c) for the EER specialization in Figure 4.4. Option 8D is used to handle overlapping subclasses by including m *Boolean* type fields, one for *each* subclass. It can also be used for disjoint classes. Each type field t_i can have a domain {yes, no}, where a value of yes indicates that the tuple is a member of subclass S_i. This option is illustrated in Figure 9.2(d) for the EER specialization in Figure 4.5, where Mflag and Pflag are the type fields. Notice that it is also possible to create a single type of m bits instead of the m type fields.

When we have a multilevel specialization (or generalization) hierarchy or lattice, we do not have to follow the same mapping option for all the specializations. Instead, we can use one mapping option for part of the hierarchy or lattice and other options for other parts. Figure 9.3 shows one possible mapping into relations for the lattice of Figure 4.7. Here we used option 8A for PERSON/ {EMPLOYEE, ALUMNUS, STUDENT}, option 8C for EMPLOYEE/{STAFF, FACULTY, STUDENT_ASSISTANT}, and option 8D for both STUDENT_ASSISTANT/{RESEARCH_ASSISTANT, TEACHING_ASSISTANT}, STUDENT/STUDENT_ASSISTANT (in STUDENT), and STUDENT/{GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. In Figure 9.3, all attributes whose names end with 'Type' or 'Flag' are type fields.

Petitioner Uber Ex-1052, 0317

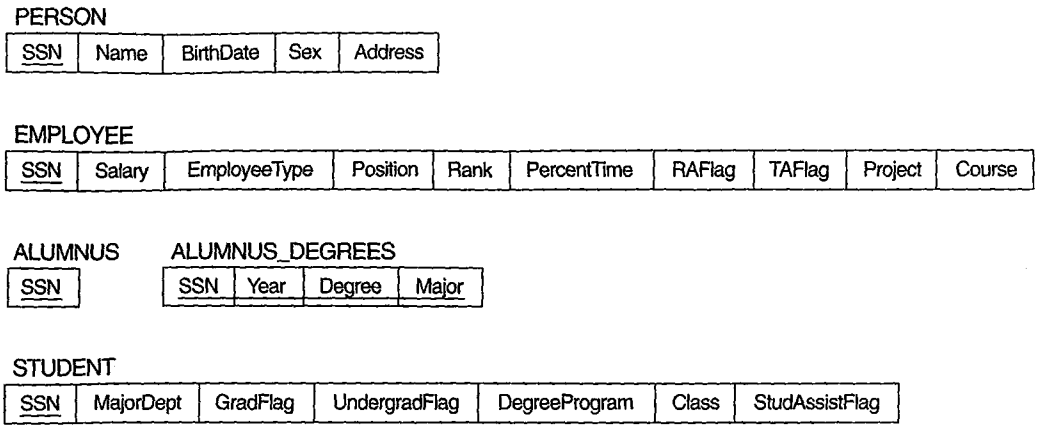


Figure 9.3 Mapping the EER specialization lattice shown in Figure 4.7 using multiple options.

9.2.2 Mapping of Shared Subclasses

A shared subclass, such as ENGINEERING_MANAGER of Figure 4.6, is a subclass of several superclasses. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category. We can apply any of the options discussed in step 8 to a shared subclass, although usually option 8A is used. In Figure 9.3, options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT in EMPLOYEE and STUDENT, respectively.

9.2.3 Mapping of Categories

A category is a subclass of the *union* of two or more superclasses that can have different keys because they can be of different entity types. An example is the OWNER category shown in Figure 4.8, which is a subset of the union of three entity types PERSON, BANK, and COMPANY. The other category in that figure, REGISTERED_VEHICLE, has two superclasses that have the same key attribute.

For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a **surrogate key,** when creating a relation to correspond to the category. This is because the keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the category. We can now create a relation schema OWNER to correspond to the OWNER category, as illustrated in Figure 9.4, and include any attributes of the category in this relation. The primary key of OWNER is the surrogate key OwnerId. We also add the surrogate key attribute OwnerId as foreign key to each relation corresponding to a superclass of the category, to specify the correspondence in values between the surrogate key and the key of each superclass.

For a category whose superclasses have the same key, such as VEHICLE in Figure 4.8, there is no need for a surrogate key. The mapping of the REGISTERED_VEHICLE category, which illustrates this case, is also shown in Figure 9.4.

The next three sections discuss some relational query languages that are important both theoretically and in practice.

Petitioner Uber Ex-1052, 0318

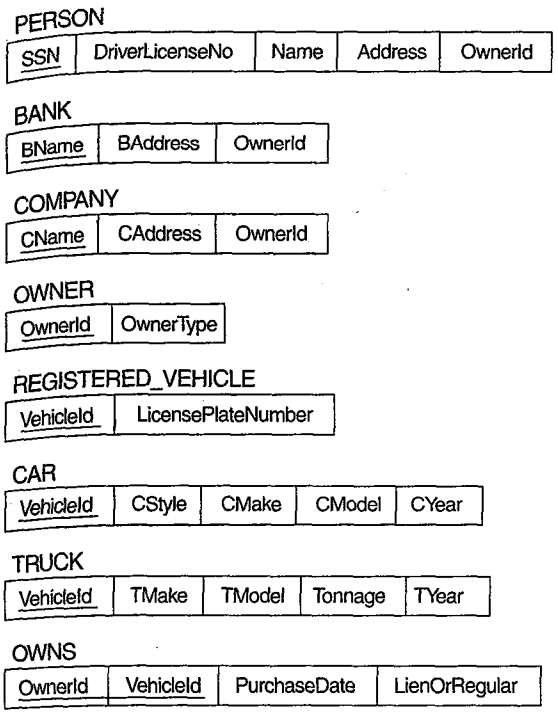


Figure 9.4 Mapping the categories of Figure 4.8 to relations.

9.3 The Tuple Relational Calculus

Relational calculus is a formal query language where we write one **declarative** expression to specify a retrieval request and hence there is no description of how to evaluate a query; a calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence it can be considered as a **procedural** way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query.

It has been shown that any retrieval that can be specified in the relational algebra can also be specified in the relational calculus, and vice versa; in other words, the **expressive power** of the two languages is *identical*. This has led to the definition of the concept of a relationally complete language. A relational query language L is considered **relationally complete** if we can express in L any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive

Petitioner Uber Ex-1052, 0319

power of high-level query languages. However, as we saw in Section 7.5, certain frequently required queries in database applications cannot be expressed in relational algebra or calculus. Most relational query languages are relationally complete but have *more expressive power* than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering. All our examples will again refer to the database shown in Figures 7.5 and 7.6. The queries are the same as those we presented in relational algebra in Chapter 7.

9.3.1 Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables.** Each tuple variable usually **ranges over** a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form

{t | COND(t)}

where t is a tuple variable and COND(t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND(t). For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

{t | EMPLOYEE(t) **and** t.SALARY>50000}

The condition EMPLOYEE(t) specifies that the *range relation* of tuple variable t is EMPLOYEE. Each EMPLOYEE tuple t that satisfies the condition t.SALARY>50000 will be retrieved. Notice that t.SALARY references attribute SALARY of tuple variable t; this notation resembles how attribute names are qualified with relation names or aliases in SQL. In the notation of Chapter 7, t.SALARY is the same as writing t[SALARY].

The above query retrieves all attribute values for each selected EMPLOYEE tuple t. To retrieve only *some* of the attributes—say, the first and last names—we write

{t.FNAME, t.LNAME | EMPLOYEE(t) **and** t.SALARY>50000}

This is equivalent to the following SQL query:

```
SELECT  T.FNAME, T.LNAME
FROM    EMPLOYEE AS T
WHERE   T.SALARY>50000 ;
```

Informally, we need to specify the following information in a tuple calculus expression:

1. For each tuple variable t, the **range relation** R of t. This value is specified by a condition of the form R(t).
2. A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.

Petitioner Uber Ex-1052, 0320

3. A set of attributes to be retrieved, the **requested attributes.** The values of these attributes are retrieved for each selected combination of tuples.

Observe the correspondence of the preceding items to a simple SQL query: item 1 corresponds to the FROM-clause relation names; item 2 corresponds to the WHERE-clause condition; and item 3 corresponds to the SELECT-clause attribute list. Before we discuss the formal syntax of tuple relational calculus, consider another query we have seen before.

QUERY 0

Retrieve the birthdate and address of the employee (or employees) whose name is 'John B. Smith'.

Q0 : {t.BDATE, t.ADDRESS | EMPLOYEE(t) **and** t.FNAME='John' **and** t.MINIT='B' **and** t.LNAME='Smith'}

In tuple relational calculus, we first specify the requested attributes t.BDATE and t.ADDRESS for each selected tuple t. Then we specify the condition for selecting a tuple following the bar (|)—namely, that t be a tuple of the EMPLOYEE relation whose FNAME, MINIT, and LNAME attribute values are 'John', 'B', and 'Smith', respectively.

9.3.2 Expressions and Formulas in Tuple Relational Calculus

A general **expression** of the tuple relational calculus is of the form

$$\{t_1.A_1, t_2.A_2, \ldots, t_n.A_n \mid \text{COND}(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

where $t_1, t_2, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula**[5] of the tuple relational calculus. A formula is made up of predicate calculus atoms, which can be one of the following:

1. An atom of the form $R(t_i)$, where R is a relation name and t_i is a tuple variable. This atom identifies the range of the tuple variable t_i as the relation whose name is R.
2. An atom of the form $t_i.A$ **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, >, \geq, <, \leq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, and B is an attribute of the relation on which t_j ranges.
3. An atom of the form $t_i.A$ **op** c or c **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, >, \geq, <, \leq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, B is an attribute of the relation on which t_j ranges, and c is a constant value.

5. Also called a **well-formed formula** or **wff** in mathematical logic.

Petitioner Uber Ex-1052, 0321

Each of the preceding atoms evaluates to either TRUE or FALSE for a specific combination of tuples; this is called the **truth value** of an atom. In general, a tuple variable ranges over all possible tuples "in the universe." For atoms of type 1, if the tuple variable is assigned a tuple that is a *member of the specified relation* R, the atom is TRUE; otherwise it is FALSE. In atoms of types 2 and 3, if the tuple variables are assigned to tuples such that the values of the specified attributes of the tuples satisfy the condition, then the atom is TRUE.

A **formula** (condition) is made up of one or more atoms connected via the logical operators **and**, **or**, and **not** and is defined recursively as follows:

1. Every atom is a formula.
2. If F_1 and F_2 are formulas, then so are (F_1 **and** F_2), (F_1 **or** F_2), **not**(F_1), and **not** (F_2). The truth values of these four formulas are derived from their component formulas F_1 and F_2 as follows:
   a. (F_1 **and** F_2) is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
   b. (F_1 **or** F_2) is FALSE if both F_1 and F_2 are FALSE; otherwise it is TRUE.
   c. **not**(F_1) is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.
   d. **not**(F_2) is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

9.3.3 The Existential and Universal Quantifiers

In addition, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** (∀) and the existential quantifier (∃). Truth values for formulas with quantifiers are described in 3 and 4 below; first, however, we need to define the concepts of free and bound tuple variables in a formula. Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an (∃ t) or (∀ t) clause; otherwise, it is free. Formally, we define a tuple variable in a formula as **free** or **bound** according to the following rules:

- An occurrence of a tuple variable in a formula F that *is an atom* is free in F.
- An occurrence of a tuple variable t is free or bound in a formula made up of logical connectives—(F_1 **and** F_2), (F_1 **or** F_2), **not**(F_1), and **not**(F_2)—depending on whether it is free or bound in F_1 or F_2 (if it occurs in either). Notice that in a formula of the form F = (F_1 **and** F_2) or F = (F_1 **or** F_2), a tuple variable may be free in F_1 and bound in F_2, or vice versa. In this case, one occurrence of the tuple variable is bound and the other is free in F.
- All *free* occurrences of a tuple variable t in F are **bound** in a formula F' of the form F' = (∃ t)(F) or F' = (∀ t)(F). The tuple variable is bound to the quantifier specified in F'. For example, consider the formulas:

  F_1 : d.DNAME='Research'

  F_2 : (∃t)(d.DNUMBER=t.DNO)

  F_3 : (∀d)(d.MGRSSN='333445555')

  The tuple variable d is free in both F_1 and F_2, whereas it is bound to the universal quantifier in F_3. Variable t is bound to the (∃) quantifier in F_2.

Petitioner Uber Ex-1052, 0322

We can now give rules 3 and 4 for the definition of a formula we started earlier:

3. If F is a formula, then so is (∃ t)(F), where t is a tuple variable. The formula (∃ t)(F) is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurrences of t in F; otherwise (∃ t)(F) is FALSE.
4. If F is a formula, then so is (∀ t)(F), where t is a tuple variable. The formula (∀ t)(F) is TRUE if the formula F evaluates to TRUE for *every tuple* (in the universe) assigned to free occurrences of t in F; otherwise (∀ t)(F) is FALSE.

The (∃) quantifier is called an existential quantifier because a formula (∃ t)(F) is true if "there exists" some tuple that makes F TRUE. For the universal quantifier, (∀ t)(F) is TRUE if every possible tuple that can be assigned to free occurrences of t in F is substituted for t, and F is TRUE for *every such substitution*. It is called the universal (or for all) quantifier because every tuple in "the universe of" tuples must make F TRUE to make the quantified formula TRUE.

9.3.4 Example Queries Using the Existential Quantifier

We will use some of the same queries shown in Chapter 7 to give a flavor of how the same queries are specified in relational algebra and in relational calculus. Notice that some queries are easier to specify in the relational algebra than in the relational calculus, and vice versa.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

Q1 : {t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) **and** (∃d)
(DEPARTMENT(d) **and** d.DNAME='Research' **and** d.DNUMBER=t.DNO) }

The *only free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (|). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified in Q1, the attributes FNAME, LNAME, and ADDRESS are retrieved for each such tuple. The conditions EMPLOYEE(t) and DEPARTMENT(d) specify the range relations for t and d. The condition d.DNAME = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra, whereas the condition d.DNUMBER = t.DNO is a **join condition** and serves a similar purpose to the JOIN operation (see Chapter 7).

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.

Petitioner Uber Ex-1052, 0323

Q2 : {p.PNUMBER, p.DNUM, m.LNAME, m.BDATE, m.ADDRESS | PROJECT(p) and
EMPLOYEE(m) and p.PLOCATION='Stafford' and
((∃d)(DEPARTMENT(d) and p.DNUM=d.DNUMBER and
d.MGRSSN=m.SSN)) }

In Q2 there are two free tuple variables, p and m. Tuple variable d is bound to the existential quantifier. The query condition is evaluated for every combination of tuples assigned to p and m; and out of all possible combinations of tuples to which p and m are bound, only the combinations that satisfy the condition are selected.

Several tuple variables in a query can range over the same relation. For example, to specify the query Q8—for each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor—we specify two tuple variables e and s that both range over the EMPLOYEE relation:

Q8 : {e.FNAME, e.LNAME, s.FNAME, s.LNAME | EMPLOYEE(e) and EMPLOYEE(s) and
e.SUPERSSN=s.SSN}

QUERY 3'

Find the name of each employee who works on *some* project controlled by department number 5. This is a variation of query 3 in which "all" is changed to "some." In this case we need two join conditions and two existential quantifiers.

Q3' : {e.LNAME, e.FNAME | EMPLOYEE(e) and ((∃ x)(∃ w)
(PROJECT(x) and WORKS_ON(w) and x.DNUM=5 and w.ESSN=e.SSN and
x.PNUMBER=w.PNO)) }

QUERY 4

Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as manager of the controlling department for the project.

Q4 : {p.PNUMBER | PROJECT(p) and
(((∃ e)(∃ w)(EMPLOYEE(e) and WORKS_ON(w) and
w.PNO=p.PNUMBER and e.LNAME='Smith' and e.SSN=w.ESSN))
or
((∃ m)(∃ d)(EMPLOYEE(m) and DEPARTMENT(d) and
p.DNUM=d.DNUMBER and d.MGRSSN=m.SSN and m.LNAME='Smith'))) }

Compare this with the relational algebra version of this query in Chapter 7. The UNION operation in relational algebra can usually be substituted with an **or** connective in relational calculus. In the next section we discuss the relationship between the universal and existential quantifiers and show how one can be transformed into the other.

Petitioner Uber Ex-1052, 0324

9.3.5 Transforming the Universal and Existential Quantifiers

We now introduce some well-known transformations from mathematical logic that relate the universal and existential quantifiers. It is possible to transform a universal quantifier into an existential quantifier, and vice versa, and to get an equivalent expression. One general transformation can be described informally as follows: transform one type of quantifier into the other with negation (preceded by **not**); **and** and **or** replace one another; a negated formula becomes unnegated; and an unnegated formula becomes negated. Some special cases of this transformation can be stated as follows:

(∀ x) (P(x)) ≡ **not** (∃ x) (**not** (P(x)))

(∃ x) (P(x)) ≡ **not** (∀ x) (**not** (P(x)))

(∀ x) (P(x) **and** Q(x)) ≡ **not** (∃ x) (**not** (P(x)) **or not** (Q(x)))

(∀ x) (P(x) **or** Q(x)) ≡ **not** (∃ x) (**not** (P(x)) **and not** (Q(x)))

(∃ x) (P(x)) **or** Q(x)) ≡ **not** (∀ x) (**not** (P(x)) **and not** (Q(x)))

(∃ x) (P(x) **and** Q(x)) ≡ **not** (∀ x) (**not** (P(x)) **or not** (Q(x)))

Notice also that the following is true, where the ⇒ symbol stands for **implies:**

(∀ x) (P(x)) ⇒ (∃ x) (P(x))

not (∃ x) (P(x)) ⇒ **not** (∀ x) (P(x))

9.3.6 Using the Universal Quantifier

Whenever we use a universal quantifier, it is quite judicious to follow a few rules to ensure that our expression makes sense. We discuss these rules with respect to Query 3.

QUERY 3

Find the names of employees who work on *all* the projects controlled by department number 5. One way of specifying this query is by using the universal quantifier as shown.

Q3 : {e.LNAME, e.FNAME | EMPLOYEE(e) **and** ((∀ x)(**not**(PROJECT(x)) **or not**(x.DNUM=5)
or ((∃ w)(WORKS_ON(w) **and** w.ESSN=e.SSN **and** x.PNUMBER=w.PNO)))) }

We can break up Q3 into its basic components as follows:

Q3 : {e.LNAME, e.FNAME | EMPLOYEE(e) **and** F' }

F' = ((∀ x)(**not**(PROJECT(x)) **or** F_1))

F_1 = **not** (x.DNUM=5) **or** F_2

F_2 = ((∃ w)(WORKS_ON(w) **and** w.ESSN = e.SSN **and** x.PNUMBER=w.PNO))

Petitioner Uber Ex-1052, 0325

We want to make sure that a selected employee e works on *all the projects* controlled by department 5, but the definition of universal quantifier says that to make the quantified formula true, the inner formula must be true *for all tuples in the universe*. The trick is to exclude from the universal quantification all tuples that we are not interested in by making the condition TRUE *for all such tuples*. This is necessary because a universally quantified tuple variable, such as x in Q3, must evaluate to TRUE *for every possible tuple* assigned to it to make the quantified formula TRUE. The first tuples to exclude are those that are not in the relation R of interest. Then we exclude the tuples we are not interested in from R itself. Finally, we specify a condition F_2 that must hold on all the remaining tuples in R. Hence, we can explain Q3 as follows:

1. For the formula F' = (∀ x)(F) to be TRUE, we must have the formula F be TRUE *for all tuples in the universe that can be assigned to x*. However, in Q3 we are only interested in F being TRUE for all tuples of the PROJECT relation that are controlled by department 5. Hence, the formula F is of the form (**not**(PROJECT(x)) **or** F_1). The '**not**(PROJECT(x)) **or** . . .' condition is TRUE for all tuples *not in the PROJECT relation* and has the effect of eliminating these tuples from consideration in the truth value of F_1. For every tuple in the project relation, F_1 must be TRUE if F' is to be TRUE.
2. Using the same line of reasoning, we do not want to consider tuples in the PROJECT relation that are not controlled by department number 5, since we are only interested in PROJECT tuples whose DNUM = 5. We can therefore write:

   if (x.DNUM=5) **then** F_2

   which is equivalent to

   (**not** (x.DNUM=5) **or** F_2)

   Formula F_1, hence, is of the form **not**(x.DNUM=5) **or** F_2. In the context of Q3, this means that, for a tuple x in the PROJECT relation, either its DNUM ≠ 5 or it must satisfy F_2.
3. Finally, F_2 gives the condition that we want to hold for a selected EMPLOYEE tuple: that the employee works on *every PROJECT tuple that has not been excluded yet*. Such employee tuples are selected by the query.

In English, Q3 gives the following condition for selecting an EMPLOYEE tuple e: for every tuple x in the PROJECT relation with x.DNUM = 5, there must exist a tuple w in WORKS_ON such that w.ESSN = e.SSN and w.PNO = x.PNUMBER. This is equivalent to saying that EMPLOYEE e works on every PROJECT x in DEPARTMENT number 5. (Whew!)

Using the general transformation from universal to existential quantifiers given in Section 9.3.5, we can rephrase the query in Q3 as shown in Q3A:

Q3A : {e.LNAME, e.FNAME | EMPLOYEE(e) **and** (**not** (∃ x) (PROJECT(x) **and** (x.DNUM=5) **and**

(**not** (∃ w)(WORKS_ON(w) **and** w.ESSN=e.SSN **and** x.PNUMBER=w.PNO))))}

We now give some additional examples of queries that use quantifiers.

Petitioner Uber Ex-1052, 0326

QUERY 6

Find the names of employees who have no dependents.

Q6 : {e.FNAME, e.LNAME | EMPLOYEE(e) **and** (**not** (∃d)(DEPENDENT(d) **and** e.SSN=d.ESSN))}

Using the general transformation rule, we can rephrase Q6 as follows:

Q6A : {e.FNAME, e.LNAME | EMPLOYEE(e) **and** ((∀ d) (**not** (DEPENDENT(d)) **or not** (e.SSN=d.ESSN)))}

QUERY 7

List the names of managers who have at least one dependent.

Q7 : {e.FNAME, e.LNAME | EMPLOYEE(e) **and** ((∃ d) (∃ p)
(DEPARTMENT(d) **and** DEPENDENT(p) **and** e.SSN=d.MGRSSN **and**
p.ESSN=e.SSN))}

The above query is handled by interpreting "managers who have at least one dependent" as "managers for whom there exists some dependent."

9.3.7 Safe Expressions

Whenever we use universal quantifiers, existential quantifiers, or negation of predicates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe.** For example, the expression

{t | **not** (EMPLOYEE(t))}

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. If we follow the rules for Q3 discussed earlier, we will get a safe expression when using universal quantifiers. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression:* This is the set of all values that either appear as constant values in the expression or exist in any tuple of the relations referenced in the expression. The domain of {t | **not**(EMPLOYEE(t))} is the set of all attribute values appearing in some tuple of the EMPLOYEE relation (for any attribute). The domain of the expression Q3A would include all values appearing in EMPLOYEE, PROJECT, and WORKS_ON (unioned with the value 5 appearing in the query itself).

An expression is said to be **safe** if all values in its result are from the domain of the expression. Notice that the result of {t | **not**(EMPLOYEE(t))} is unsafe, since it will, in general, include tuples (and hence values) from outside the EMPLOYEE relation; such values are not in the domain of the expression. All of our other examples are safe expressions.

Petitioner Uber Ex-1052, 0327

9.3.8 Quantifiers in SQL

The EXISTS function in SQL is similar to the existential quantifier of the relational calculus. When we write:

```
SELECT ...
FROM ...
WHERE EXISTS (SELECT *
              FROM   R AS X
              WHERE  P(X) )
```

in SQL, it is equivalent to saying that a tuple variable X ranging over the relation R is existentially quantified. The nested query on which the EXISTS function is applied is normally correlated with the outer query; that is, the condition P(X) includes some attribute from the outer query relations. The WHERE condition of the outer query evaluates to TRUE if the nested query returns a nonempty result that contains one or more tuples.

SQL does not include a universal quantifier. Use of a negated existential quantifier **not** ($\exists$ x) by writing NOT EXISTS is how SQL supports universal quantification, as illustrated by Q3 in Chapter 8.

9.4 The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply, **domain calculus.** The language QBE that is related to domain calculus was developed almost concurrently with SQL at IBM Research, Yorktown Heights. The formal specification of the domain calculus was proposed after the development of the QBE system.

The domain calculus differs from the tuple calculus in the *type of variables* used in formulas: rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute. An expression of the domain calculus is of the form

$$\{x_1, x_2, \ldots, x_n \mid \text{COND}(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$$

where $x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ are domain variables that range over domains (of attributes) and COND is a **condition** or **formula** of the domain relational calculus. A formula is made up of **atoms.** The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form $R(x_1, x_2, \ldots, x_j)$, where R is the name of a relation of degree j and each x_i, $1 \leq i \leq j$, is a domain variable. This atom states that a list of values of $<x_1, x_2, \ldots, x_j>$ must be a tuple in the relation whose name is R, where x_i is the value of the ith attribute value of the tuple. To make a domain calculus expression more concise, we *drop the commas* in a list of variables; thus we write

   $\{x_1, x_2, \ldots, x_n \mid R(x_1\ x_2\ x_3)$ **and** $\ldots\}$

Petitioner Uber Ex-1052, 0328

instead of:

$\{x_1, x_2, \ldots, x_n \mid R(x_1, x_2, x_3)$ **and** $\ldots\}$

2. An atom of the form x_i **op** x_j, where **op** is one of the comparison operators in the set $\{=, >, \geq, <, \leq, \neq\}$ and x_i and x_j are domain variables.
3. An atom of the form x_i **op** c or c **op** x_j, where **op** is one of the comparison operators in the set $\{=, >, \geq, <, \leq, \neq\}$, x_i and x_j are domain variables, and c is a constant value.

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the **truth values** of the atoms. In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation R, then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here. Some examples of queries specified in the domain calculus follow. We will use lowercase letters l, m, n, . . ., x, y, z for domain variables.

QUERY 0

Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.

Q0 : {uv | (∃ q) (∃ r) (∃ s) (∃ t) (∃ w) (∃ x) (∃ y) (∃ z)
(EMPLOYEE(qrstuvwxyz) **and** q='John' **and** r='B' **and** s='Smith')}

We need ten variables for the EMPLOYEE relation, one to range over the domain of each attribute in order. Of the ten variables q, r, s, . . ., z, only u and v are free. We first specify the *requested attributes*, BDATE and ADDRESS, by the domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (|)—namely, that the sequence of values assigned to the variables qrstuvwxyz be a tuple of the EMPLOYEE relation and that the values for q (FNAME), r (MINIT), and s (LNAME) be 'John', 'B', and 'Smith', respectively. For convenience, we will quantify only those variables *actually appearing in a condition* (these would be q, r, and s in Q0) in the rest of our examples.

An alternative notation for writing this query is to assign the constants 'John', 'B', and 'Smith' directly as shown in Q0A, where all variables are free:

Q0A : {uv | EMPLOYEE('John','B','Smith',t,u,v,w,x,y,z) }

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

Q1 : {qsv | (∃ z) (∃ l) (∃ m) (EMPLOYEE(qrstuvwxyz) **and**
DEPARTMENT(lmno) **and** l='Research' **and** m=z)}

Petitioner Uber Ex-1052, 0329

A condition relating two domain variables that range over attributes from two relations, such as m = z in Q1, is a **join condition;** whereas a condition that relates a domain variable to a constant, such as l = 'Research', is a **selection condition.**

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birthdate, and address.

Q2 : {iksuv | (∃ j) (∃ m)(∃ n) (∃ t)(PROJECT(hijk) **and** EMPLOYEE(qrstuvwxyz) **and** DEPARTMENT(lmno) **and** k=m **and** n=t **and** j='Stafford')}

QUERY 6

Find the names of employees who have no dependents.

Q6 : {qs | (∃ t) (EMPLOYEE(qrstuvwxyz) **and** (**not**(∃ l) (DEPENDENT(lmnop) **and** t=l)))}

Query 6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A : {qs | (∃ t) (EMPLOYEE(qrstuvwxyz) **and** ((∀ l) (**not**(DEPENDENT(lmnop)) **or** **not**(t=l)))))}

QUERY 7

List the names of managers who have at least one dependent.

Q7 : {sq | (∃ t) (∃ j) (∃ l)(EMPLOYEE(qrstuvwxyz) **and** DEPARTMENT(hijk) **and** DEPENDENT(lmnop) **and** t=j **and** l=t)}

As we mentioned earlier, it can be shown that any query that can be expressed in the relational algebra can also be expressed in the domain or tuple relational calculus. Also, any safe expression in the domain or tuple relational calculus can be expressed in the relational algebra.

9.5 Overview of the QBE Language

The Query-By-Example (QBE) language is important because it is one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the QMF (Query Management Facility) interface option to DB2. The language was also implemented in the PARADOX DBMS, and is related to a point-and-click type interface in the ACCESS DBMS (see Chapter 10). It differs from SQL in that the user does not have to specify a structured query explicitly; rather, the query is formulated by filling in **templates** of relations that are displayed on a monitor screen. Figure 9.5 shows how these templates

Petitioner Uber Ex-1052, 0330

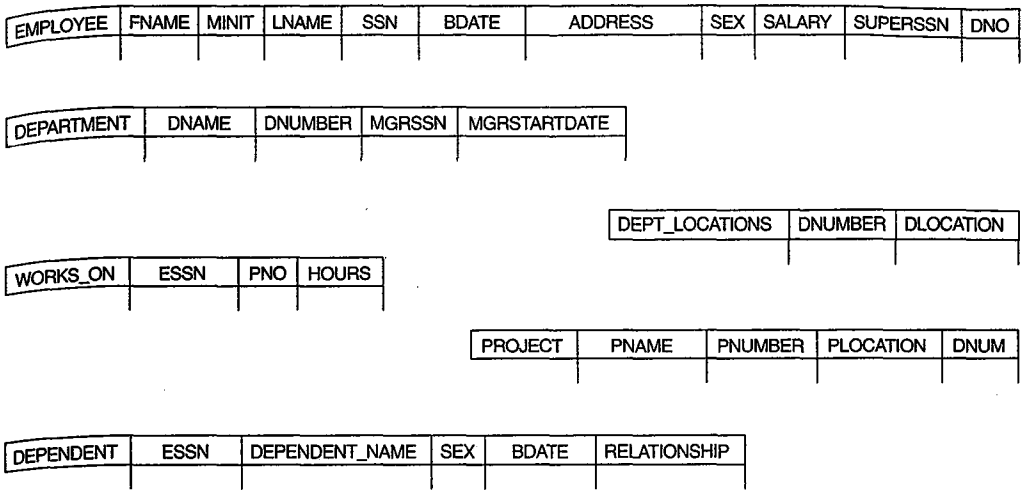


Figure 9.5 The relational schema of Figure 7.6 as it may be displayed by QBE.

may look for the database of Figure 7.6. The user does not have to remember the names of attributes or relations, because they are displayed as part of these templates. In addition, the user does not have to follow any rigid syntax rules for query specification; rather, constants and variables are entered in the columns of the templates to construct an **example** related to the retrieval or update request. QBE is related to the domain relational calculus, as we shall see, and its original specification has been shown to be relationally complete.

9.5.1 Basic Retrievals in QBE

In QBE, retrieval queries are specified by filling in one or more rows in the templates of the tables. For a single relation query, we enter either constants or **example elements** (a QBE term) in the columns of the template of that relation. An example element stands for a domain variable and is specified as an example value preceded by the underscore character (_). Additionally, a P. prefix (called the P dot operator) is entered in certain columns to indicate that we would like to print (or display) values in those columns for our result. The constants specify values that must be exactly matched in those columns.

For example, consider the query QO: "Retrieve the birthdate and address of John B. Smith." We show in Figures 9.6(a) through 9.6(d) how this query can be specified in a progressively more terse form in QBE. In Figure 9.6(a) an example of an employee is presented as the type of row that we are interested in. By leaving John B. Smith as constants in the FNAME, MINIT, and LNAME columns, we are specifying an exact match in those columns. All the rest of the columns are preceded by an underscore indicating that they are domain variables (example elements). The P. prefix is placed in the BDATE and ADDRESS columns to indicate that we would like to output value(s) in those columns.

Petitioner Uber Ex-1052, 0331

(a)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	_123456789	P._9/1/60	P._100 Main, Houston, TX	_M	_25000	_123456789	_3

(b)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith		P._9/1/60	P._100 Main, Houston, TX				

(c)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith		P._X	P._Y				

(d)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith		P.	P.				

Figure 9.6 Four ways of specifying the query Q0 in QBE.

Q0 can be abbreviated as shown in Figure 9.6(b). There is no need to specify example values for columns in which we are not interested. Moreover, because example values are completely arbitrary, we can just specify variable names for them, as shown in Figure 9.6(c). Finally, we can also leave out the example values entirely, as shown in Figure 9.6(d), and just specify a P. under the columns to be retrieved.

To see how retrieval queries in QBE are similar to the domain relational calculus, compare Figure 9.6(d) with Q0 (simplified) in domain calculus, which is as follows:

Q0 : { uv | EMPLOYEE(qrstuvwxyz) **and** q='John' **and** r='B' **and** s='Smith'}

We can think of each column in a QBE template as an *implicit domain variable*; hence, FNAME corresponds to the domain variable q, MINIT corresponds to r, . . ., and DNO corresponds to z. In the QBE query, the columns with P. correspond to variables specified to the left of the bar in domain calculus, whereas the columns with constant values correspond to tuple variables with equality selection conditions on them. The condition EMPLOYEE(qrstuvwxyz) and the existential quantifiers are implicit in the QBE query because the template corresponding to the EMPLOYEE relation is used.

In QBE, the user interface first allows the user to choose the tables (relations) needed to formulate a query by displaying a list of all relation names. The templates for the chosen relations are then displayed. The user moves to the appropriate columns in the templates and specifies the query. Special function keys were provided to move among templates and perform certain functions.

We now give examples to illustrate basic facilities of QBE. Comparison operators other than = (such as > or ≥) may be entered in a column before typing a constant value. For example, the query Q0A: "List the social security numbers of employees who work more than 20 hours per week on project number 1," can be specified as shown in Figure 9.7(a). For more complex conditions, the user can ask for a **condition box,** which is created by pressing a particular function key. The user can then type the complex condi-

Petitioner Uber Ex-1052, 0332

(a)

WORKS_ON	ESSN	PNO	HOURS
	P.		>20

(b)

WORKS_ON	ESSN	PNO	HOURS
	P.	_PX	_HX

CONDITIONS

_HX>20 AND (_PX = 1 OR _PX = 2)

(c)

WORKS_ON	ESSN	PNO	HOURS
	P.	1	>20
	P.	2	>20

Figure 9.7 Specifying complex conditions in QBE. (a) The same query Q0A. (b) The query Q0B with a condition box. (c) The query Q0B without a condition box.

tion.[6] For example, the query Q0B—"List the social security numbers of employees who work more than 20 hours per week on either project 1 or project 2"—can be specified as shown in Figure 9.7(b).

Some complex conditions can be specified without a condition box. The rule is that all conditions specified on the same row of a relation template are connected by the **and** logical connective (*all* must be satisfied by a selected tuple), whereas conditions specified on distinct rows are connected by **or** (*at least one* must be satisfied). Hence, Q0B can also be specified, as shown in Figure 9.7(c), by entering two distinct rows in the template.

Now consider query Q0C: "List the social security numbers of employees who work on *both* project 1 and project 2"; this cannot be specified as in Figure 9.8(a), which lists those who work on *either* project 1 or project 2. The example variable _ES will bind itself to ESSN values in <−, 1, −> tuples *as well as* to those in <−, 2, −> tuples. Figure 9.8(b) shows how to specify Q0C correctly, where the condition (_EX = _EY) in the box makes the _EX and _EY variables bind only to identical ESSN values.

In general, once a query is specified, the resulting values are displayed in the template under the appropriate columns. If the result contains more rows than can be displayed on the screen, most QBE implementations have function keys to allow scrolling up and down

6. Negation with the ¬ symbol is *not* allowed in a condition box.

Petitioner Uber Ex-1052, 0333

(a)

WORKS_ON	ESSN	PNO	HOURS
	P._ES	1	
	P._ES	2	

(b)

WORKS_ON	ESSN	PNO	HOURS
	P._EX	1	
	P._EY	2	

CONDITIONS
_EX = _EY

Figure 9.8 Specifying EMPLOYEES who work on both projects. (a) Incorrect specification of an AND condition. (b) Correct specification.

the rows. Similarly, if a template or several templates are too wide to appear on the screen, it is possible to scroll sideways to examine all the templates.

A join operation is specified in QBE by using the *same variable*[7] in the columns to be joined. For example, the query Q1: "List the name and address of all employees who work for the 'Research' department," can be specified as shown in Figure 9.9(a). Any number of joins can be specified in a single query. We can also specify a **result table** to display the result of the join query, as shown in Figure 9.9(a); this is needed if the result includes attributes from two or more relations. If no result table is specified, the system provides the query result in the columns of the various relations, which may make it difficult to interpret. Figure 9.9(a) also illustrates the feature of QBE for specifying that all attributes of a relation should be retrieved, by placing the P. operator under the relation name in the relation template.

To join a table with itself, we specify different variables to represent the different references to the table. For example, query Q8—"For each employee retrieve the employee's first and last name as well as the first and last name of his or her immediate supervisor"—can be specified as shown in Figure 9.9(b), where the variables starting with E refer to an employee and those starting with S refer to a supervisor.

9.5.2 Grouping, Aggregation, and Database Modification in QBE

Next, consider the types of queries that require grouping or aggregate functions. A grouping operator G. can be specified in a column to indicate that tuples should be grouped by

7. A variable is called an **example element** in QBE manuals.

Petitioner Uber Ex-1052, 0334

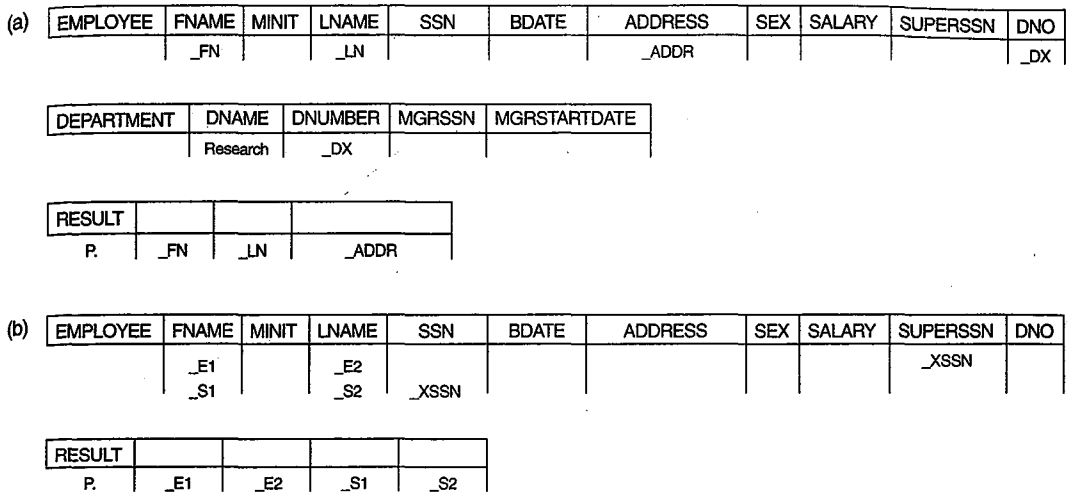


Figure 9.9 Illustrating JOIN and result relations in QBE. (a) The query Q1. (b) The query Q8.

the value of that column. Common functions can be specified, such as AVG., SUM., CNT. (count), MAX., and MIN. In QBE the functions AVG., SUM., and CNT. are applied to distinct values within a group in the default case. If we want these functions to apply to all values, we must use the prefix ALL.[8] This convention is *different* in SQL, where the default is to apply a function to all values.

Figure 9.10(a) shows query Q23, which counts the number of *distinct* salary values in the EMPLOYEE relation. Query Q23A (Figure 9.10b) counts all salary values, which is the same as counting the number of employees. Figure 9.10(c) shows Q24, which retrieves each department number and the number of employees and average salary within each department; hence, the DNO column is used for grouping as indicated by the G. function. Several of the operators G., P., and ALL can be specified in a single column. Figure 9.10(d) shows query Q26, which displays each project name and the number of employees working on it for projects on which more than two employees work.

QBE has a negation symbol, ¬, which is used in a manner similar to the NOT EXISTS function in SQL. Figure 9.11 shows query Q6, which lists the names of employees who have no dependents. The negation symbol ¬ says that we will select values of the _SX variable from the EMPLOYEE relation only if they do not occur in the DEPENDENT relation. The same effect can be produced by placing a ¬ _SX in the ESSN column.

Although the QBE language as originally proposed was shown to support the equivalent of the EXISTS and NOT EXISTS functions of SQL, the QBE implementation in QMF (under the DB2 system) does *not* provide this support. Hence, the QMF version of QBE, which we discuss here, is *not relationally complete*. Queries such as Q3—"Find employees who work on *all* projects controlled by department 5"—*cannot* be specified.

8. ALL in QBE is unrelated to the universal quantifier.

Petitioner Uber Ex-1052, 0335

(a)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
								P.CNT.		

(b)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
								P.CNT.ALL		

(c)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
				P.CNT.ALL				P.AVG.ALL		P.G.

(d)

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	P.	_PX		

WORKS_ON	ESSN	PNO	HOURS
	P.CNT._EX	G._PX	

CONDITIONS
CNT._EX>2

Figure 9.10 Functions and grouping in QBE. (a) The query Q23. (b) The query Q23A. (c) The query Q24. (d) The query Q26.

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	P.		P.	_SX						

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
¬	_SX				

Figure 9.11 Illustrating negation by the query Q6.

There are three QBE operators for modifying the database: I. for insert, D. for delete, and U. for update. The insert and delete operators are specified in the template column under the relation name, whereas the update operator is specified under the columns to be updated. Figure 9.12(a) shows how to insert a new EMPLOYEE tuple. For deletion, we first enter the D. operator and then specify the tuples to be deleted by a condition (Figure 9.12b). To update a tuple, we specify the U. operator under the attribute name, followed by the new value of the attribute. We should also select the tuple or tuples to be updated in the usual way. Figure 9.12(c) shows an update request to increase the salary of 'John Smith' by 10 percent and also to reassign him to department number 4.

QBE also has data definition capabilities. The tables of a database can be specified interactively, and a table definition can also be updated by adding, renaming, or removing a column. We can also specify various characteristics for each column, such as whether it is a key of the relation, what its data type is, and whether an index should be created on

Petitioner Uber Ex-1052, 0336

(a)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
I.	Richard	K	Marini	653298653	30-DEC-52	98 Oak Forest, Katy, TX	M	37000	987654321	4

(b)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
D.				653298653						

(c)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John		Smith					U._S*1.1		U.4

Figure 9.12 Modifying the database in QBE. (a) Insertion. (b) Deletion. (c) Update in QBE.

that field. QBE also has facilities for view definition, authorization, storing query definitions for later use, and so on.

QBE does not use the "linear" style of SQL; rather, it is a "two-dimensional" language, because users specify a query moving around the full area of the screen. Tests on users have shown that QBE is easier to learn than SQL, especially for nonspecialists. In this sense, QBE was the first user-friendly "visual" relational database language.

More recently, numerous other user-friendly interfaces have been developed for commercial database systems. The use of menus, graphics, and forms is now becoming quite common. Visual query languages, which are still not so common, are likely to be offered with commercial relational databases in the future.

9.6 Summary

This chapter covered two topics that are not directly related: relational schema design by ER-to-relational mapping and other relational languages. The reason they were grouped in one chapter is to conclude our conceptual coverage of the relational model. In Section 9.1, we showed how a conceptual schema design in the ER model can be mapped to a relational database schema. An algorithm for ER-to-relational mapping was given and illustrated by examples from the COMPANY database. Table 9.1 summarized the correspondences between the ER and relational model constructs and constraints. We then showed additional steps for mapping the constructs from the EER model into the relational model.

We then presented the basic concepts behind relational calculus, a declarative formal query language for the relational model, which is based on the branch of mathematical logic called predicate calculus. There are two types of relational calculi: (1) the tuple relational calculus, which uses tuple variables that range over tuples (rows) of relations, and (2) the domain relational calculus, which uses domain variables that range over domains (columns of relations).

In relational calculus, a query is specified in a single declarative statement, without specifying any order or method for retrieving the query result. In contrast, a relational

Petitioner Uber Ex-1052, 0337

algebra expression implicitly specifies a sequence of operations with an ordering to retrieve the result of a query. Hence, relational calculus is often considered to be a higher-level language than the relational algebra because a relational calculus expression states *what* we want to retrieve regardless of *how* the query may be executed.

We discussed the syntax of relational calculus queries using both tuple and domain variables. We also discussed the existential quantifier ($\exists$) and the universal quantifier ($\forall$). We saw that relational calculus variables are bound by these quantifiers. We saw in detail how queries with universal quantification are written, and we discussed the problem of specifying safe queries whose results are finite. We also discussed rules for transforming universal into existential quantifiers, and vice versa. It is the quantifiers that give expressive power to the relational calculus, making it equivalent to relational algebra.

The SQL language, described in Chapter 8, has its roots in the tuple relational calculus. A SELECT–PROJECT–JOIN query in SQL is similar to a tuple relational calculus expression, if we consider each relation name in the FROM clause of the SQL query to be a tuple variable with an implicit existential quantifier. The EXISTS function in SQL is equivalent to the existential quantifier and can be used in its negated form (NOT EXISTS) to specify universal quantification. There is no explicit equivalent of a universal quantifier in SQL. There is no analog to grouping and aggregation functions in relational calculus.

We then gave an overview of the QBE language, which is the first graphical query language with minimal syntax and is based on the domain relational calculus. We discussed it with several examples.

Review Questions

9.1. Discuss the correspondences between the ER model constructs and the relational model constructs. Show how each ER model construct can be mapped to the relational model, and discuss any alternative mappings. Discuss the options for mapping EER model constructs.

9.2. In what sense does relational calculus differ from relational algebra, and in what sense are they similar?

9.3. How does tuple relational calculus differ from domain relational calculus?

9.4. Discuss the meanings of the existential quantifier ($\exists$) and the universal quantifier ($\forall$).

9.5. Define the following terms with respect to the tuple calculus: *tuple variable*, *range relation*, *atom*, *formula*, *expression*.

9.6. Define the following terms with respect to the domain calculus: *domain variable*, *range relation*, *atom*, *formula*, *expression*.

9.7. What is meant by a *safe expression* in relational calculus?

9.8. When is a query language called relationally complete?

9.9. Why must the insert I. and delete D. operators of QBE appear under the relation name in a relation template, not under a column name?

9.10. Why must the update U. operators of QBE appear under a column name in a relation template, not under the relation name?

Petitioner Uber Ex-1052, 0338

Exercises

9.11. Try to map the relational schema of Figure 7.20 into an ER schema. This is part of a process known as *reverse engineering*, where a conceptual schema is created for an existing implemented database. State any assumption you make.

9.12. Figure 9.13 shows an ER schema for a database that may be used to keep track of transport ships and their locations for maritime authorities. Map this schema into a relational schema, and specify all primary keys and foreign keys.

9.13. Map the BANK ER schema of Exercise 3.23 (shown in Figure 3.17) into a relational schema. Specify all primary keys and foreign keys. Repeat for the AIRLINE schema (Figure 3.16) of Exercise 3.19 and for the other schemas for Exercises 3.16 through 3.24.

9.14. Specify queries a, b, c, e, f, i, and j of Exercise 7.18 in both the tuple relational calculus and the domain relational calculus.

9.15. Specify queries a, b, c, and d of Exercise 7.20 in both the tuple relational calculus and the domain relational calculus.

9.16. Specify queries of Exercise 8.16 in both the tuple relational calculus and the domain relational calculus. Also specify these queries in the relational algebra.

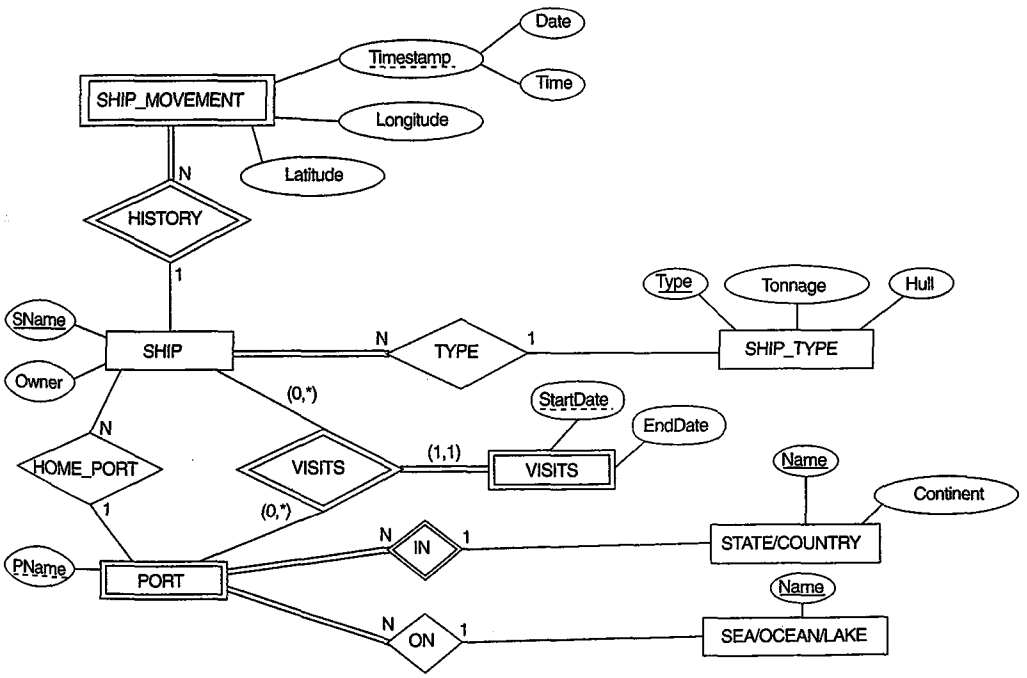


Figure 9.13 An ER schema for a SHIP_TRACKING database.

Petitioner Uber Ex-1052, 0339

9.17. In a tuple relational calculus query with n tuple variables, what would be the typical minimum number of join conditions? Why? What is the effect of having a smaller number of join conditions?

9.18. Rewrite the domain relational calculus queries that followed Q0 in Section 9.5 in the style of the abbreviated notation of Q0A, where the objective is to minimize the number of domain variables by writing constants in place of variables wherever possible.

9.19. Consider this query: Retrieve the SSNs of employees who work on at least those projects on which the employee with SSN = 123456789 works. This may be stated as (FORALL x) (IF P THEN Q), where

- x is a tuple variable that ranges over the PROJECT relation.
- P ≡ employee with SSN = 123456789 works on project x.
- Q ≡ employee e works on project x.

Express the query in tuple relational calculus, using the rules

- $(\forall x)(P(x)) \equiv \text{not}(\exists x)(\text{not}(P(x)))$.
- (IF P THEN Q) ≡ (not(P) or Q).

9.20. Show how you may specify the following relational algebra operations in both tuple and domain relational calculus.

a. $\sigma_{A=C}(R(A, B, C))$.
b. $\pi_{<A, B>}(R(A, B, C))$.
c. R(A, B, C) * S(C, D, E).
d. R(A, B, C) ∪ S(A, B, C).
e. R(A, B, C) ∩ S(A, B, C).
f. R(A, B, C) − S(A, B, C).
g. R(A, B, C) × S(D, E, F).
h. R(A, B) ÷ S(A).

9.21. Suggest extensions to the relational calculus so that it may express the following types of operations discussed in Section 6.6: (a) aggregate functions and grouping; (b) OUTER JOIN operations; (c) recursive closure queries.

9.22. Specify some of the queries of Exercises 7.18 and 8.14 in QBE.

9.23. Specify the updates of Exercise 7.19 in QBE.

9.24. Specify the queries of Exercise 8.16 in QBE.

9.25. Specify the updates of Exercise 8.17 in QBE.

9.26. Specify the queries and updates of Exercises 7.23 and 7.24 in QBE.

9.27. Map the EER diagrams in Figures 4.10 and 4.17 into relational schemas. Justify your choice of mapping options.

Selected Bibliography

Codd (1971) introduced the language ALPHA, which is based on concepts of tuple relational calculus. ALPHA also includes the notion of aggregate functions, which goes beyond relational calculus. The original formal definition of relational calculus was given by Codd (1972), which also provided an algorithm that transforms any tuple relational

Petitioner Uber Ex-1052, 0340

calculus expression to relational algebra. Codd defined relational completeness of a query language to mean at least as powerful as relational calculus. Ullman (1988) describes a formal proof of the equivalence of relational algebra with the safe expressions of tuple and domain relational calculus. Abiteboul et al. (1995) and Atzeni and deAntonellis (1993) give a detailed treatment of formal relational languages.

Although ideas of domain relational calculus were initially proposed in the QBE language (Zloof 1975), the concept was formally defined by Lacroix and Pirotte (1977). The experimental version of the Query-By-Example system is described in (Zloof 1977). The ILL language (Lacroix and Pirotte 1977a) is based on domain relational calculus. Whang et al. (1990) extends QBE with universal quantifiers. The QUEL language (Stonebraker et al. 1976) is based on tuple relational calculus, with implicit existential quantifiers but no universal quantifiers, and was implemented in the INGRES system. Thomas and Gould (1975) report the results of experiments comparing the ease of use of QBE to SQL. The commercial QBE functions are described in an IBM manual (1978), and a quick reference card is available (IBM 1978a). Appropriate DB2 reference manuals discuss the QBE implementation for that system. Visual query languages of which QBE is an example are being proposed as a means of querying databases; conferences such as the Visual Database Systems Workshop (e.g., Spaccapietra and Jain 1995) have a number of proposals for such languages.

Petitioner Uber Ex-1052, 0341

CHAPTER 10 Examples of Relational Database Management Systems: Oracle and Microsoft Access

In this chapter we turn our attention to the implementation of the relational data model in commercial systems. Because the relational database management system (RDBMS) family encompasses such a large number of products, we cannot within the scope of this book compare the features or evaluate all of them; rather, we focus in depth on two representative systems: Oracle, which is representative of the larger products that originated from mainframe computers, and Microsoft Access, a product that is appealing to the PC platform user. Our goal here will be to show how these products have a similar set of RDBMS features and functionality yet have different ways of packaging and offering them.

Section 10.1 presents a historical overview of the development of RDBMSs, and Sections 10.2 through 10.5 describe the Oracle RDBMS. Section 10.2 describes the architecture and main functions of the Oracle system. The data modeling in terms of schema objects, the languages, and the facilities of methods and triggers are presented in Section 10.3. Section 10.4 describes how Oracle organizes storage in the system. Section 10.5 presents some examples of programming in Oracle. Section 10.6 presents an overview of the tools available in Oracle for database design and application development. Later in the book we will discuss the distributed version of Oracle (Section 24.6) and in Chapter 13 we will highlight the object-relational features in Oracle 8, which extend Oracle with object-oriented features.

The Microsoft Access product presently comes bundled with Office 97 to be used on Windows and Windows NT machines. In Section 10.7 we give an overview of Microsoft Access including data definition and manipulation, and its graphic interactive facilities

Petitioner Uber Ex-1052, 0342

for ease of querying. Section 10.8 gives a summary of the features and functionality of Access related to forms, reports, and macros and briefly discusses some additional facilities available in Access.

10.1 Relational Database Management Systems: A Historical Perspective

After the relational model was introduced in 1970, there was a flurry of experimentation with relational ideas. A major research and development effort was initiated at IBM's San Jose (now called Almaden) Research Center. It led to the announcement of two commercial relational DBMS products by IBM in the 1980s: SQL/DS for DOS/VSE (disk operating system/virtual storage extended) and for VM/CMS (virtual machine/conversational monitoring system) environments, introduced in 1981; and DB2 for the MVS operating system, introduced in 1983. Another relational DBMS, INGRES, was developed at the University of California, Berkeley, in the early 1970s and commercialized by Relational Technology, Inc., in the late 1970s. INGRES became a commercial RDBMS marketed by Ingres, Inc., a subsidiary of ASK, Inc., and is presently marketed by Computer Associates. Other popular commercial RDBMSs include Oracle of Oracle, Inc.; Sybase of Sybase, Inc.; RDB of Digital Equipment Corp, now owned by Compaq; INFORMIX of Informix, Inc.; and UNIFY of Unify, Inc.

Besides the RDBMSs mentioned above, many implementations of the relational data model appeared on the personal computer (PC) platform in the 1980s. These include RIM, RBASE 5000, PARADOX, OS/2 Database Manager, DBase IV, XDB, WATCOM SQL, SQL Server (of Sybase, Inc.), SQL Server (of Microsoft), and most recently Access (also of Microsoft, Inc.). They were initially single-user systems, but more recently they have started offering the client/server database architecture (see Chapters 17 and 24) and are becoming compliant with Microsoft's *Open Database Connectivity* (ODBC), a standard that permits the use of many front-end tools with these systems.

The word *relational* is also used somewhat inappropriately by several vendors to refer to their products as a marketing gimmick. To qualify as a genuine relational DBMS, a system must have at least the following properties:[1]

1. It must store data as relations such that each column is independently identified by its column name and the ordering of rows is immaterial.
2. The operations available to the user, as well as those used internally by the system, should be true relational operations; that is, they should be able to generate new relations from old relations.
3. The system must support at least one variant of the JOIN operation.

1. Codd (1985) specified 12 rules for determining whether a DBMS is relational. Codd (1990) presents a treatise on extended relational models and systems, identifying more than 330 features of relational systems, divided into 18 categories.

Petitioner Uber Ex-1052, 0343

Although we could add to the above list, we propose these criteria as a very minimal set for testing whether a system is relational. It is easy to see that some of the so-called relational DBMSs do not satisfy these criteria.

We begin with a description of Oracle, currently one of the more widely used RDBMSs. Because some concepts in the discussion may not have been introduced yet, we will give references to later chapters in the book when necessary. Those interested in getting a deeper understanding may review the appropriate concepts in those sections and should refer to the system manuals.

10.2 The Basic Structure of the Oracle System

Traditionally, RDBMS vendors have chosen to use their own terminology in describing products in their documentation. In this section we will thus describe the organization of the Oracle system in its own nomenclature. We will try to relate this terminology to our own wherever possible. It is interesting to see how the RDBMS vendors have designed software packages that basically follow the relational model yet offer a whole variety of features needed to accomplish the design and implementation of large databases and their applications.

An **Oracle server** consists of an **Oracle database**—the collection of stored data, including log and control files—and the **Oracle Instance**—the processes, including Oracle (system) processes and user processes taken together, created for a specific instance of the database operation. Oracle server supports SQL to define and manipulate data. In addition, it has a procedural language—called PL/SQL—to control the flow of SQL, to use variables, and to provide error-handling procedures. Oracle can also be accessed through general purpose programming languages such as C or JAVA.

10.2.1 Oracle Database Structure

The Oracle database has two primary structures: (1) a *physical structure*—referring to the actual stored data—and (2) a *logical structure*—corresponding to an abstract representation of the stored data, which roughly corresponds to the conceptual schema of the database.[2] The database contains the following types of files:

- One or more *data files*; these contain the actual data.
- Two or more log files called *redo log files* (see Chapter 21 on database recovery); these record all changes made to data and are used in the process of recovering, if certain changes do not get written to permanent storage.

2. Some of the discussion in this section uses terms that have not been introduced yet. They are essential for a discussion of the complete architecture of Oracle. Readers may refer to the appropriate chapters where these terms are defined and explained.

Petitioner Uber Ex-1052, 0344

- One or more *control files;* these contain control information such as database name, file names and locations, and a database creation timestamp. This file is also needed for recovery purposes.
- *Trace files* and an *alert log;* background processes have a trace file associated with them and the alert log maintains major database events (see Chapter 23 on active databases).

Both the log file and control files may be multiplexed—that is, multiple copies may be written to multiple devices.

The structure of an Oracle database consists of the definition of the database in terms of **schema objects** and one or more **tablespaces.** The schema objects contain definitions of tables, views, sequences, stored procedures, indexes, clusters, and database links. Tablespaces, segments, and extents are the terms used to describe *physical storage structures*; they govern how the physical space of the database is used (see Section 10.4).

Oracle Instance. As we described earlier, the set of processes that constitute an instance of the server's operation is called an Oracle Instance, which consists of a System Global Area and a set of background processes. Figure 10.1 is a standard architecture diagram for Oracle, showing a number of user processes in the foreground and an Oracle process in the background. It has the following components:

- *System global area (SGA):* This area of memory is used for database information shared by users. Oracle assigns an SGA area when an instance starts. For optimal performance, the SGA is generally made as large as possible, while still fitting in real memory. The SGA in turn is divided into several types of memory structures:
  1. *Database buffer cache:* This keeps the most recently accessed data blocks from the database. By keeping most frequently accessed data blocks in this cache, the disk I/O activity can be significantly reduced.
  2. Redo log buffer, which is the buffer for the redo log file and is used for recovery purposes.
  3. Shared pool, which contains shared memory constructs; these include shared SQL areas, which contain parse trees of SQL queries and execution plans for executing SQL statements (see Chapter 18).
- *User processes:* Each user process corresponds to the execution of some application (for example, an Oracle Forms application) or some tool.
- *Program global area (PGA)* (not shown in Figure 10.1): This is a memory buffer that contains data and control information for a server process. A PGA is created by Oracle when a server process is started.
- *Oracle processes:* A process (sometimes called a job or task) is a "thread of control" or a mechanism in an operating system that can execute a series of steps. A process has its own private memory area where it runs. Oracle processes are divided into server processes and background processes. We review the types of Oracle processes and their specific functions next.

Petitioner Uber Ex-1052, 0345

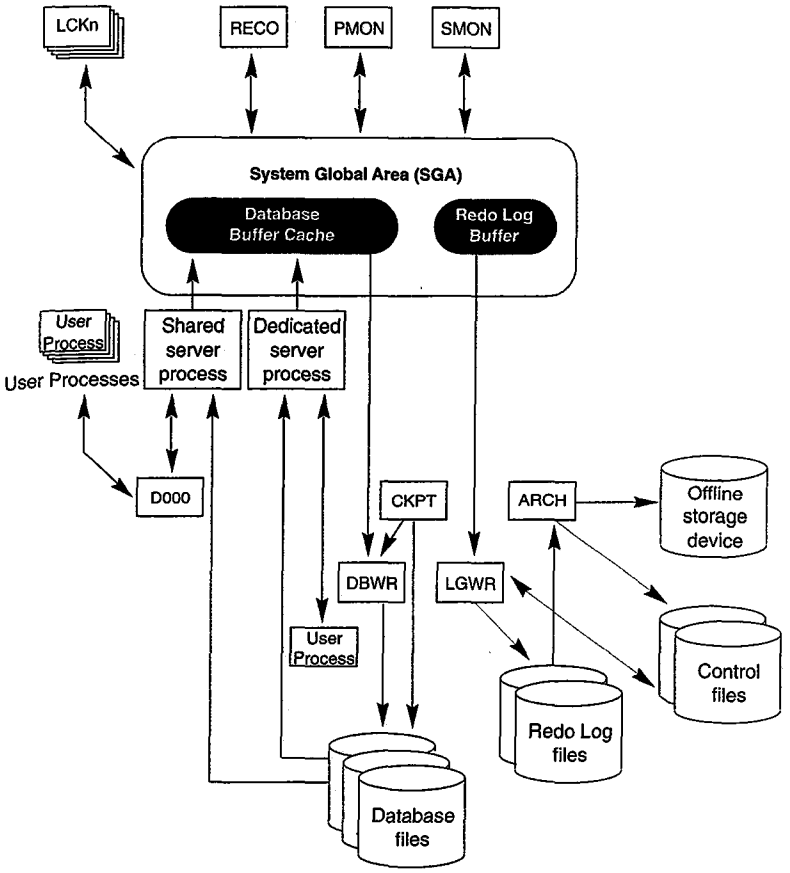


Figure 10.1 An Oracle instance. (From Oracle (1997a).)
Copyright © Oracle Corporation, 1997. All rights reserved.

10.2.2 Oracle Processes

Oracle creates **server processes** to handle requests from connected user processes. In a *dedicated server configuration*, a server process handles requests for a single user process. A more efficient alternative is a *multithreaded server configuration*, in which many user processes share a small number of server processes.

Petitioner Uber Ex-1052, 0346

The **background processes** are created for each instance of Oracle; they perform I/O asynchronously and provide parallelism for better performance and reliability. Since we have not discussed the internals of DBMSs, which we will do in Chapters 17 onward, we can only briefly describe what these background processes do; references to the appropriate chapters are included.

- *Database Writer (DBWR):* Writes the modified blocks from the buffer cache to the data files on disk. Since Oracle uses write-ahead logging (see Chapter 21), DBWR does *not* need to write blocks when a transaction commits (see Chapter 19 for definition of commit). Instead, it performs *batched writes* whenever buffers need to be freed up.
- *Log writer (LGWR):* Writes from the log buffer area to the on-line disk log file.
- *Checkpoint (CKPT):* Refers to an event at which all modified buffers in the SGA since the last checkpoint are written to the data files (see Chapter 19). The CKPT process works with DBWR to execute a checkpointing operation.
- *System monitor (SMON):* Performs instance recovery, manages storage areas by making the space contiguous, and recovers transactions skipped during recovery.
- *Process monitor (PMON):* Performs process recovery when a user process fails. It is also responsible for managing the cache and other resources used by a user process.
- *Archiver (ARCH):* Archives on-line log files to archival storage (for example, tape) if configured to do so.
- *Recoverer process (RECO):* Resolves distributed transactions that are pending due to a network or systems failure in a distributed database (see Chapter 24).
- *Dispatchers (Dnnn):* In multithreaded server configurations, route requests from connected user processes to available shared server processes. There is one dispatcher per standard communication protocol supported.
- *Lock processes (LCKn):* Used for inter-instance locking when Oracle runs in a parallel server mode.

10.2.3 Oracle Startup and Shutdown

An Oracle database is not available to users until the Oracle server has been started up and the database has been opened. Starting a database and making it available system wide requires the following steps:

1. ***Starting an instance of the database:*** The SGA is allocated and background processes are created in this step. A parameter file controlling the size of the SGA, the name of the database to which the instance can connect, etc., are set up to govern the initialization of the instance.
2. ***Mounting a database:*** This associates a previously started Oracle instance with a database. Until then it is available only to administrators. Multiple instances of Oracle may mount the same database concurrently. The database administrator chooses whether to run the database in exclusive or parallel mode. When an Oracle instance mounts a database in an *exclusive mode*, only that instance can access the database. On the other hand, if the instance is started in a *parallel or shared mode*, other instances that are started in parallel mode can also mount the database.

Petitioner Uber Ex-1052, 0347

3. ***Opening a database:*** This is a database administration activity. Opening a mounted database makes it available for normal database operations by having Oracle open the on-line data files and log files.

The reverse of the above operations will shut down an Oracle instance as follows:

1. Close the database.
2. Dismount the database.
3. Shut down the Oracle instance.

The parameter file that governs the creation of an Oracle instance contains parameters of the following types:

- Parameters that name things (for example, name of database, name and location of database's control files, names of private rollback segments[3]).
- Parameters that set limits such as maximums (for example, maximum allowable size for SGA, maximum buffer size).
- Parameters that affect capacity, called *variable parameters* (for example, the DB_BLOCK_BUFFERS parameter sets the number of data blocks to allocate in the SGA).

The database administrator may vary the parameters as part of continuous database monitoring and maintenance.

10.3 Database Structure and Its Manipulation in Oracle

Oracle was designed originally as a relational database management system (RDBMS). Starting with version 8 of the product, Oracle is being positioned as an object-relational database management system (ORDBMS). Our goal here is to review the features of Oracle including its relational and object-relational modeling facilities.[4] The main differences between Oracle 8 and the previous versions of Oracle are highlighted in Section 10.6.

10.3.1 Schema Objects

In Oracle, the term *schema* refers to a collection of data definition objects. Schema objects are the individual objects that describe tables, views, etc. There is a distinction between the logical schema objects and the physical storage components called *tablespaces*. The following schema objects are supported in Oracle. Notice that Oracle uses its own terminology that goes beyond the basic definitions of the relational model.

- *Tables:* Basic units of data that conform to the relational model discussed in Chapters 7 and 8. Each column (attribute) has a column name, datatype, and width (which depends on the type and precision).

3. For a discussion of *rollback*, see Chapter 21.

4. We will discuss object databases in Chapters 11 and 12, and object-relational systems in Chapter 13.

Petitioner Uber Ex-1052, 0348

- *Views* (see Chapter 8): Virtual tables that may be defined on base tables or on other views. If the key of the result of the join in a **join view**—that is, a view whose defining query includes a join operation—matches the key of a base table, that base table is considered **key preserved** in that view. Updating of a join view is allowed if the update applies to attributes of a base table that is key preserved. For example, consider a join of the EMPLOYEE and DEPARTMENT tables in our COMPANY database (from Figure 7.5) to yield a join view EMP_DEPT. This join table has key SSN, which matches the key of EMPLOYEE but does *not* match the key of DEPARTMENT. Hence, the EMPLOYEE base table is considered to be key preserved, but DEPARTMENT is *not*. The update on the view

```
UPDATE  EMP_DEPT
SET     Salary = Salary * 1.07
WHERE   DNO = 5 ;
```

  is acceptable because it modifies the salary attribute from the key preserved EMPLOYEE table, but the update

```
UPDATE  EMP_DEPT
SET     Mgrssn = '987654321'
WHERE   Dname = 'Research' ;
```

  fails with an error code because DEPARTMENT is not key preserved.

- *Synonyms:* Direct references to objects.[5] They are used to provide public access to an object, mask the real name or owner of an object, etc. A user may create a private synonym that is available to only that user.
- *Program units:* A function, stored procedure, or package. Procedures or functions are written in SQL or PL/SQL, which is a procedural language extension to SQL in Oracle. The term **stored procedure** is commonly used to refer to a procedure that is considered to be a part of the data definition and implements some integrity rule or business rule or a policy when it is invoked. Functions return single values. Packages provide a method of encapsulating and storing related procedures for easier management and control.
- *Sequence:* A special provision of a data type in Oracle for attribute value generation. An attribute may derive its value from a sequence, which is an automatically generated internal number. The same sequence may be used for one or more tables. As an example, an attribute EMPID for the EMPLOYEE table may be internally generated as a sequence.
- *Indexes* (see Chapter 6): An index can be generated on one or more columns of a table as requested via SQL.
- *Cluster:* A group of records from one or more tables physically stored in a mixed file (see Chapter 5). Related rows from multiple tables are physically stored together on disk blocks to improve performance.[6] By creating an **index cluster,**[7] the EMPLOYEE and DEPARTMENT tables may be clustered by the **cluster key** DNUMBER and the data is grouped

5. This is somewhat similar to naming in object databases (see Chapters 11 and 12).

6. Clustering is also often used in object databases.

7. This type of structure has also been called a *join index*, since the records to be joined together from the two files are clustered.

Petitioner Uber Ex-1052, 0349

so that the row for the DEPARTMENT with DNUMBER = 1 from the DEPARTMENT table is followed by the rows from EMPLOYEE table for all employees in that department. **Hash clusters** also group records; however, the cluster key value is hashed first, and all rows belonging to this hash value (from the different tables being clustered) are stored under the same hash bucket address.

- *Database links:* Named objects in Oracle that establish paths from one database to another. These are used in distributed databases (see Chapter 24).

10.3.2 Oracle Data Dictionary

The Oracle data dictionary is a read-only set of tables that keeps the metadata—that is, the schema description—for a database. It is composed of base tables that contain encrypted data stored by the system. User-accessible views of the dictionary decode, summarize, and conveniently display the information for users. Users are rarely given access to base tables. The special prefixes USER, ALL, and DBA are used respectively to refer to the user's view (schema objects that the user owns), expanded user view objects (objects that a user has authorization to access), and a complete set of information (for the DBA's use). We will be discussing system catalogs in detail in Chapter 17. Oracle dictionary, which is a system catalog, has the following type of information:

- Names of users.
- Security information (privileges and roles) about which users have access to what data (see Chapter 22).
- Schema objects information.
- Integrity constraints.
- Space allocation and utilization of the database objects.
- Statistics on attributes, tables, and predicates.
- Access audit trail information.

It is possible to query the data dictionary using SQL. For example, the query:

```
SELECT object_name, object_type FROM user-objects;
```

returns the information about schema objects owned by the user.

```
SELECT owner, object_name, object_type FROM all-objects;
```

returns information on all objects to which the user has access.

In addition to the above dictionary information, Oracle constantly monitors database activity and records it in tables called **dynamic performance tables.** The DBA has access to those tables to monitor system performance and may grant access to views over these tables to some users.

10.3.3 SQL in Oracle

The SQL implemented in Oracle is compliant with the SQL ANSI/ISO standard. It is similar to the SQL facilities discussed in Chapter 8 with some variations. All operations on a

Petitioner Uber Ex-1052, 0350